Java®

Библиотека профессионала

Том 2. Расширенные средства программирования

Одиннадцатое издание

Кей Хорстманн

Java®

Библиотека профессионала

Том 2. Расширенные средства программирования

Core Java®

Volume II – Advanced Features

Eleventh Edition

Cay S. Horstmann

Pearson

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam •
Cape Town Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi •
Mexico City São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Java®

Библиотека профессионала

Том 2. Расширенные средства программирования

Одиннадцатое издание

Кей Хорстманн

Диалектика

Москва • Санкт-Петербург
2020

ББК 32.973.26-018.2.75
Х82
УДК 004.432.2

ООО "Диалектика"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция *И.В. Берштейна*
По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Хорстманн, Кей С.

Х82 Java. Библиотека профессионала, том 2. Расширенные средства программирования, 11-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2020. — 864 с. : ил. — Парал. тит. англ.

SBN 978-5-907144-38-5 (рус., том 2)
ISBN 978-5-907144-30-9 (рус., многотом)

ББК 32.973.26-018.2.75

Научно-популярное издание
Кей С. Хорстманн

Java. Библиотека профессионала, том 2
Расширенные средства программирования
11-е издание

Подписано в печать 27.11.2019. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 69,66. Уч.-изд. л. 45,4.
Тираж 300 экз. Заказ № 10558.

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

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848
ISBN 978-5-907144-38-5 (рус., том 2)
ISBN 978-5-907144-30-9 (рус., многотом)
ISBN 978-0-13-516631-4 (англ.)

Оглавление

Содержание

Предисловие

К читателю

Книга, которую вы держите в руках, является вторым томом одиннадцатого издания, полностью обновленного по версии Java 11. В первом томе рассматривались основные языковые средства Java, а в этом томе речь пойдет о расширенных функциональных возможностях, которые могут понадобиться программисту для разработки программного обеспечения на высоком профессиональном уровне. Поэтому этот том, как, впрочем, и первый том настоящего и предыдущих изданий данной книги, нацелен на тех программистов, которые собираются применять технологию Java в работе над реальными проектами.

Краткий обзор книги

В целом главы этого тома составлены независимо друг от друга. Это дает читателю возможность начинать изучение материала с той темы, которая интересует его больше всего, и вообще читать главы второго тома в любом удобном ему порядке.

В **главе 1** рассматривается библиотека потоков данных в Java, придающая современные черты обработке данных благодаря тому, что программисту достаточно указать, что именно ему требуется, не вдаваясь в подробности, как получить желаемый результат. Такой подход позволяет уделить в библиотеке потоков данных основное внимание оптимальной эволюционной стратегии, которая дает особые преимущества при оптимизации параллельных вычислений.

Глава 2 посвящена организации ввода-вывода. В языке Java весь ввод-вывод осуществляется через так называемые *потоки ввода-вывода* (не путать с потоками данных, рассматриваемыми в главе 1). Такие потоки позволяют единообразно обмениваться данными между различными источниками, включая файлы, сетевые соединения и блоки памяти. В начале этой главы приводится подробное описание классов чтения и записи в потоки ввода-вывода, упрощающие обработку данных в Юникоде. Далее в ней рассматривается внутренний механизм сериализации объектов, который делает простым и удобным сохранение и загрузку объектов. И в завершение главы обсуждаются регулярные выражения, а также особенности манипулирования файлами и путями к ним. На протяжении всей

этой главы будут представлены долгожданные усовершенствования системы ввода-вывода в последних версиях Java.

Основной темой **главы 3** является XML. В ней показывается, каким образом осуществляется синтаксический анализ XML-файлов, формируется XML-разметка и выполняются XSL-преобразования. В качестве примера демонстрируется разметка компоновки Swing-формы в формате XML. В этой главе рассматривается также прикладной интерфейс API XPath, значительно упрощающий поиск мелких подробностей в больших объемах данных формата XML.

В **главе 4** рассматривается прикладной интерфейс API для работы в сети. В языке Java чрезвычайно просто решаются сложные задачи сетевого программирования. В этой главе показывается, как устанавливаются сетевые соединения с серверами, реализуются собственные серверы и организуется связь по сетевому протоколу HTTP. Здесь также описывается новый HTTP-клиент.

Глава 5 посвящена программированию баз данных. Основное внимание в ней уделяется JDBC — прикладному интерфейсу для организации доступа к базам данных из приложений на Java, который позволяет прикладным программам на Java устанавливать связь с реляционными базами данных. В этой главе также показывается, как писать полезные программы для выполнения рутинных операций с настоящими базами данных, применяя только самые основные средства интерфейса JDBC. (Для рассмотрения всех средств интерфейса JDBC потребовалась бы отдельная книга почти такого же объема, как и эта.) И в завершение главы приводятся краткие сведения об интерфейсе JNDI (Java Naming and Directory Interface — интерфейс именования и каталогов Java) и протоколе LDAP (Lightweight Directory Access Protocol — упрощенный протокол доступа к каталогам).

Ранее в библиотеках Java были предприняты две безуспешные попытки организовать обработку даты и времени. Третья попытка была успешно предпринята в версии Java 8. Поэтому в **главе 6** поясняется, как преодолевать трудности организации календарей и оперирования часовыми поясами, используя новую библиотеку даты и времени.

В **главе 7** обсуждаются вопросы интернационализации, важность которой, на наш взгляд, будет со временем только возрастать. Java относится к тем немногочисленным языкам программирования, где с самого начала предусматривалась возможность обработки данных в Юникоде, но поддержка интернационализации в Java этим не ограничивается. В частности, интернационализация прикладных программ на Java позволяет сделать их независимыми не только от платформы, но и от страны применения. В качестве примера в этой главе демонстрируется, как написать прикладную программу для расчета времени выхода на пенсию с выбором английского, немецкого или китайского языка.

В **главе 8** описываются три разные методики обработки исходного кода. Так, прикладные интерфейсы API для сценариев и компилятора дают возможность вызывать в программе на Java код, написанный на каком-нибудь языке сценариев, например JavaScript или Groovy, и компилировать его в код Java. Аннотации позволяют вводить в программу на Java произвольную информацию (иногда еще называемую *метаданными*). В этой главе показывается, каким образом обработчики аннотаций собирают аннотации на уровне источника и на уровне файлов классов и как с помощью аннотаций оказывается воздействие на поведение классов во время

выполнения. Аннотации выгодно использовать только вместе с подходящими инструментальными средствами, и мы надеемся, что материал этой главы поможет читателю научиться выбирать именно те средства обработки аннотаций, которые в наибольшей степени отвечают его потребностям.

В **главе 9** описывается модульная система на платформе Java, внедренная в версии Java 9 для того, чтобы способствовать нормальной эволюции самой платформы и базовых библиотек Java. Эта модульная система обеспечивает инкапсуляцию пакетов и предоставляет механизм для описания требований к модулям. В этой главе рассматриваются свойства модулей, на основании которых вы можете решить, стоит ли применять модули в ваших приложениях. Но даже если вы решите не применять их, вы все равно должны знать новые правила модуляризации, чтобы взаимодействовать с платформой Java и другими библиотеками, имеющими модульную организацию.

В **главе 10** представлена модель безопасности Java. Платформа Java с самого начала разрабатывалась с учетом безопасности, и в этой главе объясняется, что именно позволяет ей обеспечивать безопасность. Сначала в ней демонстрируется, как создавать свои собственные загрузчики классов и диспетчеры защиты для специальных приложений. Затем рассматривается прикладной интерфейс API для безопасности, который позволяет оснащать приложения важными средствами вроде механизма цифровых подписей сообщений и кода, а также авторизации, аутентификации и шифрования. И завершается глава демонстрацией примеров, в которых применяются такие алгоритмы шифрования, как AES и RSA.

В **главе 11** представлен весь материал по библиотеке Swing, который не вошел в первый том данной книги, в том числе описание важных и сложных компонентов деревьев и таблиц. Здесь также рассматривается прикладной интерфейс Java 2D API, которым можно пользоваться для воспроизведения реалистичных графических изображений и спецэффектов. Безусловно, разрабатывать пользовательские интерфейсы на основе библиотеки Swing приходится немногим программистам, и поэтому в этой главе особое внимание уделяется тем функциональным средствам, с помощью которых можно формировать изображения на сервере.

Глава 12 посвящена платформенно-ориентированным методам, которые позволяют вызывать функции, специально написанные для конкретной платформы, например Microsoft Windows. Очевидно, что данное языковое средство является спорным, ведь применение платформенно-ориентированных методов сводит на нет все межплатформенные преимущества Java. Тем не менее всякий, серьезно занимающийся разработкой на Java приложений для конкретных платформ, должен знать и уметь пользоваться платформенно-ориентированными средствами. Ведь иногда возникают ситуации, когда требуется обращаться к прикладному интерфейсу API операционной системы целевой платформы для взаимодействия с устройствами или службами, которые не поддерживаются на платформе Java. В этой главе показано, как это сделать, на примере организации доступа из программы на Java к прикладному интерфейсу API системного реестра Windows.

Как обычно, все главы второго тома были полностью обновлены по самой последней версии Java. Весь устаревший материал был изъят, а новые прикладные интерфейсы API, появившиеся в версиях Java 9–11, подробно рассматриваются в соответствующих местах.

Условные обозначения

Как это принято во многих компьютерных книгах, моноширинный шрифт используется для представления исходного кода.

Этой пиктограммой выделяются примечания.

Этой пиктограммой выделяются советы.

Этой пиктограммой выделяются предупреждения о потенциальной опасности.

В этой книге имеется немало примечаний к синтаксису C++, где разъясняются отличия между языками Java и C++. Можете пропустить их, если вас *не* интересует программирование на C++.

Язык Java сопровождается огромной библиотекой в виде прикладного интерфейса (API). При упоминании вызова какого-нибудь метода из прикладного интерфейса API в первый раз в конце соответствующего раздела приводится его краткое описание. Эти описания не слишком информативны, но, как мы надеемся, более содержательны, чем те, которые представлены в официальной оперативно доступной документации на прикладной интерфейс API. Имена интерфейсов выделены *полужирным*, как это делается в официальной документации. А число после имени класса, интерфейса или метода обозначает версию JDK, в которой данное средство было внедрено, как показано ниже.

Название прикладного интерфейса `1.2`

Программы с доступным исходным кодом организованы в виде примеров, как показано ниже.

Листинг 1.1. Исходный код из файла **`ScriptTest.java`**

Примеры исходного кода

Все примеры исходного кода, приведенные в этом томе в частности и в данной книге вообще, доступны в архивированном виде на посвященном ей веб-сайте по адресу `http://horstmann.com/corejava`.

Благодарности

Написание книги всегда требует значительных усилий, а ее переписывание не намного легче, особенно если учесть постоянные изменения в технологии Java. Чтобы сделать книгу полезной, необходимы совместные усилия многих

преданных людей, и автор с удовольствием выражает признательность всем, кто внес свой посильный вклад в общее дело.

Большое число сотрудников издательства Pearson оказали неоценимую помощь, хотя и остались в тени. Я хотел бы выразить им свою признательность за их усилия. Как всегда, самой горячей благодарности заслуживает мой редактор Грег Доенч — за сопровождение книги на протяжении всего процесса ее написания и издания, а также за то, что он позволил мне пребывать в блаженном неведении относительно многих скрытых деталей этого процесса. Я благодарен Джули Нахил за оказанную помощь в подготовке книги к изданию, а также Дмитрию и Алине Кирсановым — за литературное редактирование и набор рукописи книги.

Выражаю большую признательность многим читателям прежних изданий, которые сообщали о найденных ошибках и внесли массу ценных предложений по улучшению книги. Я особенно благодарен блестящему коллективу рецензентов, которые тщательно просмотрели рукопись книги, устранив в ней немало досадных ошибок.

Среди рецензентов этого и предыдущих изданий хотелось бы отметить Чака Аллисона (выпускающего редактора издания *C/C++ Users Journal*), Ланса Андерсона из компании Oracle, Алека Битона из PointBase, Inc., Клиффа Берга, Джошуа Блоха, Дэвида Брауна, Корки Картрайта, Френка Коена из PushToTest, Криса Крейна из devXsolution, доктора Николаса Дж. Де Лилло из Манхеттенского колледжа, Ракеша Дхупара из компании Oracle, Роберта Эванса, ведущего специалиста из лаборатории прикладной физики университета имени Джонса Хопкинса, Дэвида Джири из Sabreware, Джима Гиша из Oracle, Брайана Гоетца из Oracle, Анджелу Гордон, Дэна Гордона, Роба Гордона, Джона Грэя из Хартфордского университета, Камерона Грегори (`olabs.com`), Стива Хейнеса, Марти Холла из лаборатории прикладной физики в университете имени Джона Хопкинса, Винсента Харди из Adobe Systems, Дэна Харки из университета штата Калифорния в Сан-Хосе, Вильяма Хиггинса из IBM, Владимира Ивановича из PointBase, Джерри Джексона из CA Technologies, Тима Киммета из Preview Systems, Криса Лаффра, Чарли Лаи, Анжелику Лангер, Дуга Лэнгстона, Ханг Лау из университета имени Макгилла, Марка Лоуренса, Дуга Ли из SUNY Oswego, Грегори Лонгшора, Боба Линча из Lynch Associates, Филиппа Милна, консультанта, Марка Моррисси из научно-исследовательского института штата Орегон, Махеша Нилаканта из Атлантического университета штата Флорида, Хао Фам, Пола Филиона, Блейка Рагсдейла, Ильбера Рамадани из университета имени Райерсона, Стюарта Реджеса из университета штата Аризона, Саймона Риттера, Рича Розена из Interactive Data Corporation, Питера Сандерса из университета ЭССИ (ESSI), г. Ницца, Франция, доктора Пола Сангеру из университета штата Калифорния в Сан-Хосе и колледжа имени Брукса, Поля Севинка из Teamup AG, Деванг Ша, Йокиси Сабата, Ричарда Сливчака из Исследовательского центра имени Гленна, НАСА, Бредли А. Смита, Стивена Стелтинга, Кристофера Тэйлора, Люка Тэйлора из Valtech, Джорджа Тхируватукала, Кима Топли, автора книги *Core JFC, Second Edition*, Джанет Трауб, Пола Тайма, консультанта, Кристиана Улленбоома, Питера Ван Дер Линдена, Берта Уолша, Джо Уанга из Oracle и Дана Ксю из Oracle.

Кей Хорстманн, Сан-Франциско, декабрь 2018 г.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. 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 в наш адрес.

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

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

Наши электронные адреса:

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

ГЛАВА 1

Потоки данных

В этой главе...

- От итерации к потоковым операциям
- Создание потока данных
- Методы `filter()`, `map()` и `flatMap()`
- Извлечение подпотоков и объединение потоков данных
- Другие операции преобразования потоков данных
- Простые методы сведения
- Тип `Optional`
- Накопление результатов
- Накопление результатов в отображениях
- Группирование и разделение
- Нисходящие коллекторы
- Операции сведения
- Потоки данных примитивных типов
- Параллельные потоки данных

В сравнении с коллекциями потоки данных обеспечивают представление данных, позволяющее указать вычисления на более высоком концептуальном уровне, чем коллекции. С помощью потока данных можно указать, что и как именно требуется сделать с данными, а планирование операций предоставить конкретной реализации. Допустим, требуется вычислить среднее некоторого свойства. С этой целью указывается источник данных и свойство, а средствами библиотеки потоков данных можно оптимизировать вычисление, используя, например, несколько потоков исполнения для расчета сумм, подсчета и объединения результатов.

В этой главе поясняется, как пользоваться библиотекой потоков данных, которая была внедрена в версии Java 8, для обработки коллекций по принципу "что, а не как делать".

1.1. От итерации к потоковым операциям

Для обработки коллекции обычно требуется перебрать ее элементы и выполнить над ними некоторую операцию. Допустим, требуется подсчитать все длинные слова в книге. Сначала организуем их вывод списком следующим образом:

```
var contents = new String(Files.readAllBytes(
  Paths.get("alice.txt")), StandardCharsets.UTF_8);
    // прочитать текст из файла в символьную строку
List<String> words = Arrays.asList(contents.split("\\PL+"));
    // разбить полученную символьную строку на слова;
    // небуквенные символы считаются разделителями
```

А теперь можно перебрать слова таким образом:

```
int count = 0;
for (String w : words) {
  if (w.length() > 12) count++;
}
```

Ниже показано, как аналогичная операция осуществляется с помощью потоков данных.

```
long count = words.stream()
                  .filter(w -> w.length() > 12)
                  .count();
```

В последнем случае не нужно искать в цикле наглядного подтверждения операций фильтрации и подсчета слов. Сами имена методов свидетельствуют о том, что именно предполагается сделать в коде. Более того, если в цикле во всех подробностях предписывается порядок выполнения операций, то в потоке данных операции можно планировать как угодно, при условии, что будет достигнут правильный результат.

Достаточно заменить метод `stream()` на метод `parallelStream()`, чтобы организовать средствами библиотеки потоков данных параллельное выполнение операций фильтрации и подсчета слов, как показано ниже.

```
long count = words.parallelStream()
                  .filter(w -> w.length() > 12)
                  .count();
```

Потоки данных действуют по принципу "что, а не как делать". В рассматриваемом здесь примере кода мы описываем, что нужно сделать: получить длинные слова и подсчитать их. При этом мы не указываем, в каком порядке или потоке исполнения это должно произойти. Напротив, в упомянутом выше цикле точно указывается порядок организации вычислений, а следовательно, исключается всякая возможность для оптимизации.

На первый взгляд поток данных похож на коллекцию, поскольку он позволяет преобразовывать и извлекать данные. Но у потока данных имеются следующие существенные отличия.

1. Поток данных не сохраняет свои элементы. Они могут храниться в основной коллекции или формироваться по требованию.
2. Потоковые операции не изменяют их источник. Например, метод `filter()` не удаляет элементы из нового потока данных, но выдает новый поток, в котором они отсутствуют.
3. Потоковые операции выполняются *по требованию*, когда это возможно. Это означает, что они не выполняются до тех пор, пока не потребуется их результат. Так, если требуется подсчитать только пять длинных слов вместо всех слов, метод `filter()` прекратит фильтрацию после пятого совпадения. Следовательно, потоки данных могут быть бесконечными!

Вернемся к предыдущему примеру, чтобы рассмотреть его подробнее. Методы `stream()` и `parallelStream()` выдают *поток данных* для списка слов `words`. А метод `filter()` возвращает другой поток данных, содержащий только те слова, длина которых больше 12 букв. И, наконец, метод `count()` сводит этот поток данных в конечный результат.

Такая последовательность операций весьма характерна для манипулирования потоками данных. Конвейер операций организуется в следующие три стадии.

1. Создание потока данных.
2. Указание *промежуточных операций* для преобразования исходного потока данных в другие потоки — возможно, в несколько этапов.
3. Выполнение *оконечной операции* для получения результата. Эта операция принуждает к выполнению по требованию тех операций, которые ей предшествуют. А впоследствии поток данных может больше не понадобиться.

В примере кода из листинга 1.1 поток данных создается методом `stream()` или `parallelStream()`. Метод `filter()` преобразует его, а метод `count()` выполняет оконечную операцию.

Листинг 1.1. Исходный код из файла **`streams/CountLongWords.java`**

```
1  package streams;
2
3  /**
4   * @version 1.01 2018-05-01
5   * @author Cay Horstmann
6   */
7
8  import java.io.*;
9  import java.nio.charset.*;
10 import java.nio.file.*;
11 import java.util.*;
12
13 public class CountLongWords
14 {
15   public static void main(String[] args)
16         throws IOException
17   {
18     var contents = new String(Files.readAllBytes(
```

```
19          Paths.get("../gutenberg/alice30.txt")),
20                  StandardCharsets.UTF_8);
21      List<String> words = List.of(contents.split("\\PL+"));
22
23      long count = 0;
24      for (String w : words)
25      {
26        if (w.length() > 12) count++;
27      }
28      System.out.println(count);
29
30      count = words.stream()
31                  .filter(w -> w.length() > 12).count();
32      System.out.println(count);
33
34      count = words.parallelStream()
35                  .filter(w -> w.length() > 12).count();
36      System.out.println(count);
37    }
38 }
```

В следующем разделе будет показано, как создается поток данных. В трех последующих разделах рассматриваются потоковые операции преобразования, а в пяти следующих за ними разделах — оконечные операции.

java.util.stream.Stream<T> 8

- **Stream<T> filter(Predicate<? super T> *p*)**
  Возвращает поток данных, содержащий все его элементы, совпадающие с указанным предикатом ***p***.
- **long count()**
  Возвращает количество элементов в исходном потоке данных. Это оконечная операция.

java.util.Collection<E> 1.2

- **default Stream<E> stream()**
- **default Stream<E> parallelStream()**
  Возвращают последовательный или параллельный поток данных, состоящий из элементов исходной коллекции.

1.2. Создание потока данных

Как было показано выше, любую коллекцию можно преобразовать в поток данных методом `stream()` из интерфейса `Collection`. Если же вместо коллекции имеется массив, то для этой цели служит метод `Stream.of()`:

```
Stream<String> words = Stream.of(contents.split("\\PL+"));
    // Метод split() возвращает массив типа String[]
```

У метода of() имеются аргументы переменной длины, и поэтому поток данных можно построить из любого количества аргументов, как показано ниже. А для создания потока данных из части массива служит метод Arrays.stream(array, from, to).

```
Stream<String> song = Stream.of("gently", "down",
                                "the", "stream");
```

Чтобы создать поток данных без элементов, достаточно вызвать статический метод Stream.empty() следующим образом:

```
Stream<String> silence = Stream.empty();
    // Обобщенный тип <String> выводится автоматически;
    // что равнозначно вызову Stream.<String>empty()
```

Для создания бесконечных потоков данных в интерфейсе Stream имеются два статических метода. В частности, метод generate() принимает функцию без аргументов (а формально — объект функционального интерфейса Supplier<T>). Всякий раз, когда требуется потоковое значение, эта функция вызывается для получения данного значения. Например, поток постоянных значений можно получить следующим образом:

```
Stream<String> echos = Stream.generate(() -> "Echo");
```

а поток случайных чисел таким образом:

```
Stream<Double> randoms = Stream.generate(Math::random);
```

Для получения бесконечных последовательностей вроде 0 1 2 3 ... служит метод iterate(). Этот метод принимает начальное значение и функцию (а формально — объект функционального интерфейса UnaryOperator<T>) и повторно применяет функцию к предыдущему результату, как показано в следующем примере кода:

```
Stream<BigInteger> integers = Stream.iterate(
       BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```

Первым элементом такой последовательности является начальное значение BigInteger.ZERO; вторым ее элементом — значение, получаемое в результате вызова функции f(seed), или 1 (как крупное целочисленное значение); следующим элементом — значение, получаемое в результате вызова функции f(f(seed)), или 2 и т.д., где seed — начальное значение.

А для того чтобы создать конечный поток данных, достаточно ввести предикат, в котором указывается момент, когда должна быть завершена итерация, как выделено ниже полужирным. Поток данных завершится, как только предикат отклонит итеративно формируемое значение.

```
var limit = new BigInteger("10000000");
Stream<BigInteger> integers = Stream.iterate(
                   BigInteger.ZERO,
                   n -> n.compareTo(limit) < 0,
                   n -> n.add(BigInteger.ONE));
```

И, наконец, метод Stream.ofNullable() создает очень короткий поток данных из объекта. Такой поток имеет нулевую длину, если исходный объект оказывается пустым (null), а иначе — единичную длину, т.е. он содержит лишь

данный объект. Этим методом удобнее всего пользоваться вместе с методом `flatMap()`, как демонстрируется в примере, приведенном далее в разделе 1.7.7.

НА ЗАМЕТКУ! В прикладном интерфейсе Java API имеется целый ряд методов, возвращающих потоки данных. Так, в классе **`Pattern`** имеется метод **`splitAsStream()`**, разделяющий последовательность символов типа **`CharSequence`** по регулярному выражению. Например, для разделения символьной строки на отдельные слова можно воспользоваться следующим оператором:

```
Stream<String> words = Pattern.compile("\\PL+")
                                .splitAsStream(contents);
```

А статический метод **`Files.lines()`** возвращает поток данных типа **`Stream`**, содержащий все строки из файла, как показано ниже.

```
try (Stream<String> lines = Files.lines(path)) {
  Обработать строки
}
```

НА ЗАМЕТКУ! Если имеется итерируемый объект, не являющийся коллекцией, его можно преобразовать в поток данных, сделав следующий вызов:

```
StreamSupport.stream(iterable.spliterator(), false);
```

А если имеется итератор и результаты его применения требуется направить в поток данных, в таком случае можно сделать приведенный ниже вызов.

```
StreamSupport.stream(Spliterators.spliteratorUnknownSize(
                    iterator, Spliterator.ORDERED), false);
```

ВНИМАНИЕ! Во время выполнения операции над потоком данных очень важно не модифицировать коллекцию, поддерживающую этот поток. При этом следует помнить, что потоки не накапливают свои данные, которые всегда хранятся в отдельной коллекции. Если же модифицировать коллекцию при выполнении операций над потоком данных, то их результаты окажутся неопределенными. В документации на комплект JDK такое требование к коллекциям называется *невмешательством*.

Точнее говоря, коллекцию можно модифицировать до того момента, когда начнется выполнение конечной операции, поскольку промежуточные операции над потоками данных выполняются по требованию. Например, приведенный ниже фрагмент кода окажется вполне работоспособным, хотя поступать подобным образом все же не рекомендуется.

```
List<String> wordList = . . .;
Stream<String> words = wordList.stream();
wordList.add("END");
long n = words.distinct().count();
```

А следующий фрагмент кода ошибочен:

```
Stream<String> words = wordList.stream();
words.forEach(s -> if (s.length() < 12) wordList.remove(s));
    // ОШИБКА из-за вмешательства!
```

В примере кода из листинга 1.2 демонстрируются различные способы создания потока данных.

Листинг 1.2. Исходный код из файла **streams/CreatingStreams.java**

```
1  package streams;
2
3  /**
4   * @version 1.01 2018-05-01
5   * @author Cay Horstmann
6   */
7
8  import java.io.IOException;
9  import java.math.BigInteger;
10 import java.nio.charset.StandardCharsets;
11 import java.nio.file.*;
12 import java.util.*;
13 import java.util.regex.Pattern;
14 import java.util.stream.*;
15
16 public class CreatingStreams
17 {
18   public static <T> void show(String title,
19                               Stream<T> stream)
20   {
21     final int SIZE = 10;
22     List<T> firstElements = stream
23             .limit(SIZE + 1)
24             .collect(Collectors.toList());
25     System.out.print(title + ": ");
26     for (int i = 0; i < firstElements.size(); i++)
27     {
28       if (i > 0) System.out.print(", ");
29       if (i < SIZE)
30         System.out.print(firstElements.get(i));
31       else System.out.print("...");
32     }
33     System.out.println();
34   }
35
36   public static void main(String[] args)
37         throws IOException
38   {
39     Path path = Paths.get("../gutenberg/alice30.txt");
40     var contents = new String(Files.readAllBytes(path),
41                               StandardCharsets.UTF_8);
42
43     Stream<String> words =
44                     Stream.of(contents.split("\\PL+"));
45     show("words", words);
46     Stream<String> song = Stream.of("gently", "down",
47                                     "the", "stream");
48     show("song", song);
49     Stream<String> silence = Stream.empty();
50     show("silence", silence);
51
```

```
52      Stream<String> echos =
53                      Stream.generate(() -> "Echo");
54      show("echos", echos);
55
56      Stream<Double> randoms =
57                      Stream.generate(Math::random);
58      show("randoms", randoms);
59
60      Stream<BigInteger> integers = Stream.iterate(
61                          BigInteger.ONE,
62                          n -> n.add(BigInteger.ONE));
63      show("integers", integers);
64
65      Stream<String> wordsAnotherWay =
66         Pattern.compile("\\PL+").splitAsStream(contents);
67
68      show("wordsAnotherWay", wordsAnotherWay);
69
70      try (Stream<String> lines =
71           Files.lines(path, StandardCharsets.UTF_8))
72      {
73        show("lines", lines);
74      }
75
76      Iterable<Path> iterable =
77           FileSystems.getDefault().getRootDirectories();
78      Stream<Path> rootDirectories = StreamSupport.stream(
79                  iterable.spliterator(), false);
80      show("rootDirectories", rootDirectories);
81
82      Iterator<Path> iterator =
83            Paths.get("/usr/share/dict/words").iterator();
84      Stream<Path> pathComponents = StreamSupport.stream(
85             Spliterators.spliteratorUnknownSize(iterator,
86                             Spliterator.ORDERED), false);
87      show("pathComponents", pathComponents);
88    }
89 }
```

***java.util.stream.Stream* 8**

- **static <T> Stream<T> of(T... *values*)**
  Возвращает поток данных, элементами которого являются заданные значения.
- **static <T> Stream<T> empty()**
  Возвращает поток данных без элементов.
- **static <T> Stream<T> generate(Supplier<T> *s*)**
  Возвращает бесконечный поток данных, элементы которого составляются путем повторного вызова функции ***s*()**.

`java.util.stream.Stream` 8 *(окончание)*

- **`static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)`**
- **`static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> f)`**

  Возвращают бесконечный поток данных, элементы которого содержат начальные значения ***`seed`***. Заданная функция ***`f()`*** сначала вызывается с начальным значением ***`seed`***, а затем со значением предыдущего элемента и т.д. Первый из этих методов возвращает бесконечный поток данных. А поток данных, возвращаемый вторым методом, завершается, как только первому элементу не удастся выполнить предикат ***`hasNext`***.
- **`static <T> Stream<T> ofNullable(T t)`** 9

  Возвращает пустой поток данных, если объект, задаваемый в качестве параметра ***`t`***, оказывается пустым (**`null`**), а иначе — поток данных, содержащий заданный объект ***`t`***.

`java.util.Spliterators` 8

- **`static <T> Spliterator<T> spliteratorUnknownSize(Iterator<? extends T> iterator, int characteristics)`**

  Преобразует обычный итератор в разделяемый итератор неизвестного размера с характеристиками (параметр ***`characteristics`***), задаваемыми в виде комбинации битов, содержащей такие константы, как **`Spliterator.ORDERED`**.

`java.util.Arrays` 1.2

- **`static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive)`** 8

  Возвращает поток данных, элементы которого сформированы из заданного диапазона в указанном массиве.

`java.util.regex.Pattern` 1.4

- **`Stream<String> splitAsStream(CharSequence input)`** 8

  Возвращает поток данных, элементы которого являются частями входной последовательности символов (параметр ***`input`***), разделяемых по данному шаблону.

`java.nio.file.Files` 7

- **`static Stream<String> lines(Path path)`** 8
- **`static Stream<String> lines(Path path, Charset cs)`** 8

  Возвращают поток данных, элементы которого составляют строки из указанного файла в кодировке UTF-8 или в заданном наборе символов.

java.util.stream.StreamSupport 8

- **static <T> Stream<T> stream(Spliterator<T> *spliterator*, boolean *parallel*)**
  Возвращает поток данных, содержащий значения, производимые заданным разделяемым итератором.

java.lang.Iterable 5

- **Spliterator<T> spliterator()** 8
  Возвращает разделяемый итератор для данного итерируемого объекта. В реализации данного метода по умолчанию итератор не разделяется и его размер не возвращается.

java.util.Scanner 5

- **public Stream<String> tokens()** 9
  Возвращает поток символьных строк, вызывая метод **next()** из данного потока сканирования.

java.util.function.Supplier<T> 8

- **T get()**
  Возвращает получаемое значение.

1.3. Методы filter(), map() и flatMap()

В результате преобразования потока данных получается другой поток данных, элементы которого являются производными от элементов исходного потока. Ранее демонстрировалось преобразование методом `filter()`, в результате которого получается новый поток данных с элементами, удовлетворяющими определенному условию. В приведенном ниже примере кода поток символьных строк преобразуется в другой поток, содержащий только длинные слова. В качестве аргумента метода `filter()` указывается объект типа `Predicate<T>`, т.е. функция, преобразующая тип `T` в логический тип `boolean`.

```
List<String> words = ...;
Stream<String> longWords =
               words.stream().filter(w -> w.length() > 12);
```

Нередко значения в потоке данных требуется каким-то образом преобразовать. Для этой цели можно воспользоваться методом `map()`, передав ему функцию, которая и выполняет нужное преобразование. Например, буквы во всех словах можно сделать строчными следующим образом:

```
Stream<String> lowercaseWords =
               words.stream().map(String::toLowerCase);
```

В данном примере методу `map()` была передана ссылка на метод. Но вместо нее нередко передается лямбда-выражение, как показано ниже. Получающийся в итоге поток данных содержит первую букву каждого слова.

```
Stream<String> firstLetters =
               words.stream().map(s -> s.substring(0, 1));
```

При вызове метода `map()` передаваемая ему функция применяется к каждому элементу потока данных, в результате чего образуется новый поток данных с полученными результатами. А теперь допустим, что имеется метод, возвращающий не одно значение, а поток значений. В качестве примера ниже приведен метод, преобразующий символьную строку в поток символьных строк, а точнее — отдельных кодовых точек.

```
public static Stream<String> codePoints(String s)
{
  var result = new ArrayList<String>();
  int i = 0;
  while (i < s.length())
  {
    int j = s.offsetByCodePoints(i, 1);
    result.add(s.substring(i, j));
    i = j;
  }
  return result.stream();
}
```

Данный метод правильно обрабатывает символы в Юникоде, требующие двух значений типа `char`, поскольку именно так это и следует делать. Хотя вникать в такие подробности совсем не обязательно. Например, в результате вызова `codePoints("boat")` образуется поток данных `["b", "o", "a", "t"]`.

А теперь допустим, что метод `codePoints()` передается методу `map()` для преобразования потока символьных строк следующим образом:

```
Stream<Stream<String>> result =
               words.stream().map(w -> codePoints(w));
```

В итоге получится поток потоков вроде `[... ["y", "o", "u", "r"], ["b", "o", "a", "t"], ...]`. Чтобы свести его к потоку букв `[... "y", "o", "u", "r", "b", "o", "a", "t", ...]`, вместо метода `map()` следует вызвать метод `flatMap()` таким образом:

```
Stream<String> flatResult =
       words.stream().flatMap(w -> codePoints(w))
    // Вызывает метод codePoints() для каждого слова
    // и сводит результаты
```

НА ЗАМЕТКУ! Аналогичный метод **`flatMap()`** можно обнаружить и в других классах, а не только в тех, которые представляют потоки данных. Это общий принцип вычислительной техники. Допустим, имеется обобщенный тип **`G`** (например, **`Stream`**) и функции **`f()`** и **`g()`**, преобразующие некоторый тип **`T`** в тип **`G<U>`**, а тип **`U`** — в тип **`G<V>`** соответственно. В таком случае эти функции можно составить вместе, используя метод **`flatMap()`**, т.е. применить сначала функцию **`f()`**, а затем функцию **`g()`**. В этом состоит главная идея теории *монад*. Впрочем, метод **`flatMap()`** можно применять, и не зная ничего о монадах.

java.util.stream.Stream 8

- **Stream<T> filter(Predicate<? super T> *predicate*)**
  Возвращает поток данных, элементы которого совпадают с указанным предикатом.
- **<R> Stream<R> map(Function<? super T,? extends R> *mapper*)**
  Возвращает поток данных, содержащий результаты применения функции ***mapper*()** к элементам исходного потока данных.
- **<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> *mapper*)**
  Возвращает поток данных, получаемый сцеплением результатов применения функции ***mapper*()** к элементам исходного потока данных. (Следует, однако, иметь в виду, что каждый результат представляет собой отдельный поток данных.)

1.4. Извлечение подпотоков и объединение потоков данных

В результате вызова *поток*.limit(n) возвращается поток данных, оканчивающийся после n элементов или по завершении исходного потока данных, если тот оказывается более коротким. Метод limit() особенно удобен для ограничения бесконечных потоков данных до определенной длины. Так, в следующей строке кода получается поток данных, состоящий из 100 произвольных чисел:

```
Stream<Double> randoms =
               Stream.generate(Math::random).limit(100);
```

В результате вызова *поток*.skip(n) происходит совершенно противоположное: отбрасываются первые n элементов. Если вернуться к рассмотренному ранее примеру чтения текста книги, то в силу особенностей работы метода split() первым элементом потока данных оказывается нежелательная пустая строка. От нее можно избавиться, вызвав метод skip() следующим образом:

```
Stream<String> words =
               Stream.of(contents.split("\\PL+")).skip(1);
```

При вызове *поток*.takeWhile(*предикат*) из потока данных извлекаются все элементы до тех пор, пока параметр *предикат* принимает логическое значение true, после чего данный процесс останавливается.

Допустим, что метод codePoints(), упоминавшийся в предыдущем разделе, применяется для разбиения символьной строки на отдельные символы и при этом требуется собрать все первоначальные числа из натурального ряда. Этой цели можно добиться с помощью метода takeWhile() следующим образом:

```
Stream<String> initialDigits = codePoints(str).takeWhile(
               s -> "0123456789".contains(s));
```

Метод dropWhile() делает совершенно противоположное, пропуская элементы в потоке до тех пор, пока заданное условие остается истинным, формируя поток элементов до тех пор, пока это условие не станет ложным. Например:

```
Stream<String> withoutInitialWhiteSpace =
    codePoints(str).dropWhile(s -> s.trim().length() == 0);
```

Два потока данных можно соединить вместе с помощью статического метода `concat()` из интерфейса `Stream`, как показано ниже. Разумеется, первый из этих потоков не должен быть бесконечным, иначе второй поток вообще не сможет соединиться с ним.

```
Stream<String> combined = Stream.concat(
                codePoints("Hello"), codePoints("World"));
    // В итоге получается следующий поток данных:
    // ["H", "e", "l", "l", "o", "W", "o", "r", "l", "d"]
```

java.util.stream.Stream 8

- **`Stream<T> limit(long maxSize)`**

  Возвращает поток данных, состоящий из элементов исходного потока данных вплоть до заданной длины ***maxSize***.

- **`Stream<T> skip(long n)`**

  Возвращает поток данных, все элементы которого, кроме начальных ***n*** элементов, взяты из исходного потока данных.

- **`Stream<T> takeWhile(Predicate<? super T> predicate)`** 9

  Возвращает поток данных, состоящий из тех элементов данного потока, которые удовлетворяют заданному предикату.

- **`Stream<T> dropWhile(Predicate<? super T> predicate)`** 9

  Возвращает поток данных, состоящий из тех элементов данного потока, которые *не* удовлетворяют заданному предикату.

- **`static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)`**

  Возвращает поток данных, элементы которого последовательно составлены из элементов потока ***a*** и элементов потока ***b***.

1.5. Другие операции преобразования потоков данных

Метод `distinct()` возвращает поток данных, получающий свои элементы из исходного потока данных в том же самом порядке, за исключением того, что дубликаты в нем подавляются и совсем не обязательно должны быть смежными.

```
Stream<String> uniqueWords =
       Stream.of("merrily", "merrily", "merrily", "gently")
             .distinct();
    // В итоге возвращается только одна строка "merrily"
```

Для сортировки потоков данных имеется несколько вариантов метода `sorted()`. Один из них служит для обработки потоков данных, состоящих из элементов типа `Comparable`, а другой принимает в качестве параметра компаратор типа `Comparator`. В следующем примере кода символьные строки сортируются таким образом, чтобы первой в потоке данных следовала самая длинная строка:

```
Stream<String> longestFirst = words.stream().sorted(
       Comparator.comparing(String::length).reversed());
```

Как и во всех остальных операциях преобразования потоков данных, метод `sorted()` выдает новый поток данных, элементы которого берутся из исходного потока и располагаются в отсортированном порядке. Разумеется, коллекцию можно отсортировать, не прибегая к потокам данных. Метод `sorted()` удобно применять в том случае, если процесс сортировки является частью поточного конвейера.

Наконец, метод `peek()` выдает другой поток данных с теми же самыми элементами, что и у исходного потока, но передаваемая ему функция вызывается всякий раз, когда извлекается элемент. Это удобно для целей отладки, как показано ниже.

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
        .peek(e -> System.out.println("Fetching " + e))
        .limit(20).toArray();
```

Сообщение выводится в тот момент, когда элемент доступен в потоке данных. Подобным образом можно проверить, что бесконечный поток данных, возвращаемый методом `iterate()`, обрабатывается по требованию.

СОВЕТ. Если отладчик применяется для отладки кода, вычисляющего поток данных, то в теле метода, вызываемого в одной из операций преобразования, можно установить точку прерывания. В большинстве интегрированных сред разработки точки прерывания могут быть установлены и в лямбда-выражениях. Если же требуется лишь выяснить, что именно происходит в конкретной точке поточного конвейера, для этого достаточно ввести приведенный ниже фрагмент кода, установив точку прерывания во второй его строке.

```
.peek(x -> {
  return; })
```

java.util.stream.Stream 8

- **`Stream<T> distinct()`**

  Возвращает поток данных, состоящий из неповторяющихся элементов исходного потока.
- **`Stream<T> sorted()`**
- **`Stream<T> sorted(Comparator<? super T>` *`comparator`*`)`**

  Возвращают поток данных, состоящий из отсортированных элементов исходного потока. Первый метод требует, чтобы элементы были экземплярами класса, реализующего интерфейс **`Comparable`**.
- **`Stream<T> peek(Consumer<? super T>` *`action`*`)`**

  Возвращает поток данных, состоящий из тех же элементов, что и у исходного потока, передавая каждый элемент указанной функции ***`action`*`()`** по мере употребления этого элемента.

1.6. Простые методы сведения

Теперь, когда было показано, каким образом осуществляется создание и преобразование потоков данных, мы наконец-то добрались до самого главного — получения ответов на запросы данных из потоков. В этом разделе рассматриваются так называемые *методы сведения*. Они выполняют *оконечные операции*, сводя поток данных к непотоковому значению, которое может быть далее использовано

в программе. Ранее уже демонстрировался простой метод сведения `count()`, возвращающий количество элементов в потоке данных.

К числу других простых методов сведения относятся методы `max()` и `min()`, возвращающие наибольшее и наименьшее значения соответственно. Но не все так просто, поскольку эти методы на самом деле возвращают значение типа `Optional<T>`, которое заключает в себе ответ на запрос данных из потока или обозначает, что запрашиваемые данные отсутствуют, поскольку поток оказался пустым. Раньше в подобных случаях возвращалось пустое значение `null`. Но это могло привести к исключениям в связи с пустыми указателями в не полностью протестированной программе. Пользоваться типом `Optional` удобнее для обозначения отсутствующего возвращаемого значения. Более подробно тип `Optional` рассматривается в следующем разделе, а ниже показано, как получить максимальное значение из потока данных.

```
Optional<String> largest =
        words.max(String::compareToIgnoreCase);
System.out.println("largest: " + largest.getOrElse(""));
```

Метод `findFirst()` возвращает первое значение из непустой коллекции. Зачастую он применяется вместе с методом `filter()`. Так, в следующем примере кода обнаруживается первое слово, начинающееся с буквы `Q`:

```
Optional<String> startsWithQ =
        words.filter(s -> s.startsWith("Q")).findFirst();
```

Если же требуется любое совпадение, а не только первое, то следует воспользоваться методом `findAny()`, как показано ниже. Это оказывается эффективным при распараллеливании потока данных, поскольку поток может известить о любом обнаруженном в нем совпадении, вместо того чтобы ограничиваться только первым совпадением.

```
Optional<String> startsWithQ = words.parallel().filter(
                s -> s.startsWith("Q")).findAny();
```

Если же требуется лишь выяснить, имеется ли вообще совпадение, то следует воспользоваться методом `anyMatch()`, как показано ниже. Этот метод принимает предикатный аргумент, поэтому ему не требуется метод `filter()`.

```
boolean aWordStartsWithQ =
        words.parallel().anyMatch(s -> s.startsWith("Q"));
```

Имеются также методы `allMatch()` и `noneMatch()`, возвращающие логическое значение `true`, если с предикатом совпадают все элементы в потоке данных или не совпадает ни один из его элементов соответственно. Эти методы также выгодно выполнять в параллельном режиме.

java.util.stream.Stream 8

- **Optional<T> max(Comparator<? super T> *comparator*)**
- **Optional<T> min(Comparator<? super T> *comparator*)**

  Возвращают максимальный или минимальных элемент из исходного потока данных, используя порядок расположения, который определяет заданный ***comparator***, или же пустое значение типа **Optional**, если исходный поток данных пуст. Это оконечные операции.

java.util.stream.Stream 8 *(окончание)*

- **Optional<T> findFirst()**
- **Optional<T> findAny()**

  Возвращают первый или любой элемент из исходного потока данных или же значение типа **Optional**, если исходный поток данных пуст. Это оконечные операции.
- **boolean anyMatch(Predicate<? super T> *predicate*)**
- **boolean allMatch(Predicate<? super T> *predicate*)**
- **boolean noneMatch(Predicate<? super T> *predicate*)**

  Возвращают логическое значение **true**, если с заданным предикатом совпадают любые или все элементы исходного потока данных или же не совпадает ни один из его элементов.

1.7. Тип Optional

Объект типа `Optional<T>` служит оболочкой для объекта обобщенного типа `T` или же ни для одного из объектов. В первом случае считается, что значение *присутствует*. Тип `Optional<T>` служит в качестве более надежной альтернативы ссылке на обобщенный тип `T`, которая делается на объект или оказывается пустой. Но этот тип надежнее, если правильно им пользоваться. В следующем разделе поясняется, как это делается.

1.7.1. Получение необязательных значений

Для эффективного применения типа `Optional` самое главное — выбрать метод, который возвращает *альтернативный вариант*, если значение отсутствует, или *употребляет значение*, если только оно присутствует. Рассмотрим первую методику обращения с необязательными значениями. Нередко имеется значение, возможно, пустая строка "", которое требуется использовать по умолчанию в отсутствие совпадения:

```
String result = optionalString.orElse("");
    // Заключенная в оболочку строка,
    // а в ее отсутствие - пустая строка ""
```

Кроме того, можно вызвать функцию для вычисления значения по умолчанию следующим образом:

```
String result = optionalString.orElseGet(() ->
                System.getProperty("user.dir"));
    // Функция вызывается только по мере надобности
```

С другой стороны, в отсутствие значения можно сгенерировать исключение таким образом:

```
// предоставить метод, возвращающий объект исключения:
String result = optionalString.orElseThrow(
                                IllegalStateException::new);
```

java.util.Optional 8

- **T orElse(T *other*)**
  Возвращает имеющееся значение типа **Optional** или другое значение ***other***, если присутствующее значение типа **Optional** оказывается пустым.
- **T orElseGet(Supplier<? extends T> *other*)**
  Возвращает присутствующее значение типа **Optional** или результат вызова функции ***other*()**, если присутствующее значение типа **Optional** оказывается пустым.
- <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)
  Возвращает имеющееся значение типа **Optional**, или выдает результат вызова **exception Supplier** если **Optional** пуст.

1.7.2. Употребление необязательных значений

Как было показано в предыдущем разделе, если значение отсутствует, можно получить его альтернативный вариант. Другая методика обработки необязательных значений состоит в том, чтобы употребить значение только в том случае, если оно присутствует.

Метод `ifPresent()` принимает функцию в качестве аргумента, как показано ниже. Если необязательное значение существует, оно передается данной функции. В противном случае ничего не происходит.

```
optionalValue.ifPresent(v -> Обработать v);
```

Так, если требуется ввести значение во множество, при условии, что оно существует, достаточно сделать следующий вызов:

```
optionalValue.ifPresent(v -> results.add(v));
```

или просто

```
optionalValue.ifPresent(results::add);
```

Если требуется предпринять одно действие, когда необязательное значение присутствует, и другое действие, когда оно отсутствует, в таком случае можно вызвать метод `ifPresentOrElse()`:

```
optionalValue.ifPresentOrElse(
        v -> System.out.println("Found " + v),
        () -> logger.warning("No match"));
```

java.util.Optional 8

- **void ifPresent(Consumer<? super T> *consumer*)**
  Передает присутствующее значение типа **Optional** функции ***consumer*()**, если это значение оказывается непустым.
- **<U> Optional<U> map(Function<? super T,? extends U> *mapper*)**
  Возвращает результат передачи присутствующего значения типа **Optional** функции ***mapper*()**, если это значение оказывается непустым и результат не равен **null**, а иначе — пустое значение типа **Optional**.

1.7.3. Конвейеризация необязательных значений

В предыдущих разделах было показано, как получить необязательное значение из объекта типа `Optional`. Еще одна полезная методика состоит в том, чтобы вообще не затрагивать объект типа `Optional`. Необязательное значение можно преобразовать в объекте типа `Optional`, используя метод `map()`, как показано ниже. Так, если объект `optionalString` пустой, то пустым окажется и объект `transformed`.

```
Optional<String> transformed =
                 optionalString.map(String::toUpperCase);
```

В еще одном примере результат вводится в список, если он присутствует. А если объект `optionalValue` пуст, то ничего не происходит.

```
optionalValue.map(results::add);
```

НА ЗАМЕТКУ! В данном случае метод **`map()`** действует аналогично методу **`map()`** из интерфейса **`Stream`**, упоминавшемуся ранее в разделе 1.3. Если представить необязательное значение как поток данных нулевого или единичного размера, то в результате преобразования будет также получен поток данных нулевого или единичного размера, и в последнем случае применяется функция преобразования.

Аналогично с помощью метода `filter()` можно отобрать до или после преобразования только те необязательные значения, которые удовлетворяют определенному критерию. И если критерий не удовлетворяется, то на выходе из конвейера получается пустой результат.

```
Optional<String> transformed = optionalString
                 .filter(s -> s.length() >= 8)
                 .map(String::toUpperCase);
```

Вместо пустого необязательного значения можно подставить альтернативное необязательное значение, используя метод `or()`. При этом альтернативное значение вычисляется по требованию.

```
Optional<String> result = optionalString.
         or(() -> // предоставить необязательное значение
               alternatives.stream().findFirst());
```

Если у объекта `optionalString` имеется значение, то переменной `result` присваивается ссылка на этот объект. В противном случае вычисляется лямбда-выражение, и его результат присваивается переменной `result`.

`java.util.Optional` 8

- **`<U> Optional<U> map(Function<? super T,? extends U> mapper)`**

  Возвращает объект типа **`Optional`**, значение которого получается в результате применения функции ***`mapper`*`()`** к имеющемуся значению данного объекта типа **`Optional`**, а иначе — пустой объект типа **`Optional`**.

- **`Optional<T> filter(Predicate<? super T> predicate)`**

  Возвращает объект типа **`Optional`** со значением из данного объекта типа **`Optional`**, если оно удовлетворяет заданному предикату (параметр ***`predicate`***), а иначе — пустой объект типа **`Optional`**.

java.util.Optional 8 *(окончание)*

- **Optional<T> or(Supplier<? extends Optional<? extends T>> *supplier*)** 9
  Возвращает объект типа **Optional**, если он не пустой, а иначе — объект типа **Optional**, получаемый поставщиком.

1.7.4. Как не следует обрабатывать необязательные значения

Если необязательные значения типа `Optional` не применяются правильно, то они не дают никаких преимуществ по сравнению с прежним подходом, предоставлявшим выбор между чем-то существующим или несуществующим, т.е. `null`. Метод `get()` получает заключенный в оболочку элемент значения типа `Optional`, если это значение существует, а иначе — генерирует исключение типа `NoSuchElementException`. Таким образом, следующий фрагмент кода:

```
Optional<T> optionalValue = ...;
optionalValue.get().someMethod()
```

не надежнее, чем такой код:

```
T value = ...;
value.someMethod();
```

Метод `isPresent()` извещает, содержит ли значение объект типа `Optional<T>`. Но следующее выражение:

```
if (optionalValue.isPresent()) optionalValue
                  .get().someMethod();
```

не проще, чем такое:

```
if (value != null) value.someMethod();
```

НА ЗАМЕТКУ! В версии Java 10 был внедрен менее привлекательный аналог метода **get()**. Чтобы добиться явного генерирования исключения типа **NoSuchElementException**, если объект типа **optionalValue** окажется пустым, достаточно сделать вызов **optionalValue.orElseThrow()**. Такой метод был внедрен в надежде, что программисты будут вызывать его лишь в том случае, если они совершенно уверены, что объект **Optional** вообще не пустой.

Ниже приведен ряд дополнительных рекомендаций относительно надлежащего пользования типом данных `Optional`.

- Переменная типа `Optional` *вообще* не должна быть пустой (`null`).
- Не пользуйтесь полями типа `Optional`, поскольку для этого потребуется дополнительный объект. А для обозначения отсутствующего в классе поля лучше воспользоваться пустым значением `null`.
- Не размещайте объекты типа `Optional` в множестве и не пользуйтесь ими в качестве ключей к отображению. Вместо этого храните значения в отображении.

java.util.Optional 8

- **T get()**
- **T orElseThrow() 10**

  Возвращают значение данного объекта типа **Optional**, а если он пустой — генерируют исключение типа **NoSuchElementException**.
- **boolean isPresent()**

  Возвращает логическое значение **true**, если данный объект типа **Optional** не пустой.

1.7.5. Формирование необязательных значений

До сих пор обсуждалось, как употреблять объект типа `Optional`, созданный кем-то другим. Если же требуется написать метод, создающий объект типа `Optional`, то для этой цели имеется несколько статических методов. В приведенном ниже примере кода демонстрируется применение двух таких методов: `Optional.of(result)` и `Optional.empty()`.

```
public static Optional<Double> inverse(Double x) {
  return x == 0 ? Optional.empty() : Optional.of(1 / x);
}
```

Метод `ofNullable()` служит в качестве моста между возможными пустыми (`null`) и необязательными (`Optional`) значениями. Так, при вызове метода `Optional.ofNullable(obj)` возвращается результат вызова метода `Optional.of(obj)`, если объект `obj` не является пустым (`null`), а иначе — результат вызова метода `Optional.empty()`.

java.util.Optional 8

- **static <T> Optional<T> of(T *value*)**
- **static <T> Optional<T> ofNullable(T *value*)**

  Возвращают объект типа **Optional** с заданным значением. Если заданное значение ***value*** равно **null**, то первый метод генерирует исключение типа **NullPointerException**, а второй метод возвращает пустой объект типа **Optional**.
- **static <T> Optional<T> empty()**

  Возвращает пустой объект типа **Optional**.

1.7.6. Сочетание функций необязательных значений с методом `flatMap()`

Допустим, имеется метод `f()`, возвращающий объект типа `Optional<T>`, а у целевого типа `T` — метод `g()`, возвращающий объект типа `Optional<U>`. Если бы это были обычные методы, их можно было бы составить в вызов `s.f().g()`. Но такое сочетание не годится, поскольку результат вызова `s.f()` относится к типу `Optional<T>`, а не к типу `T`. Вместо этого придется сделать следующий вызов:

```
Optional<U> result = s.f().flatMap(T::g);
```

Если объект, получаемый в результате вызова `s.f()`, присутствует, то к нему применяется метод `g()`. В противном случае возвращается пустой объект типа `Optional<U>`.

Очевидно, что данный процесс можно повторить, если имеются другие методы или лямбда-выражения, возвращающие необязательные значения типа `Optional`. В таком случае из них можно составить конвейер, связав их вызовы в цепочку с методом `flatMap()`, который будет успешно завершен, если завершатся все остальные части конвейера.

В качестве примера рассмотрим надежный метод `inverse()` из предыдущего раздела. Допустим, имеется также следующий надежный метод для извлечения квадратного корня:

```
public static Optional<Double> squareRoot(Double x) {
  return x < 0 ? Optional.empty()
               : Optional.of(Math.sqrt(x));
}
```

В таком случае извлечь квадратный корень из значения, возвращаемого методом `inverse()`, можно следующим образом:

```
Optional<Double> result =
        inverse(x).flatMap(MyMath::squareRoot);
```

или таким способом, если он предпочтительнее:

```
Optional<Double> result = Optional.of(-4.0)
       .flatMap(Demo::inverse).flatMap(Demo::squareRoot);
```

Если метод `inverse()` или `squareRoot()` возвратит результат вызова метода `Optional.empty()`, то конечный результат окажется пустым.

НА ЗАМЕТКУ! Как было показано в разделе 1.3, метод **`flatMap()`** из интерфейса **`Stream`** служит для составления двух других методов, получающих потоки данных, сводя их в результирующий поток потоков. Аналогичным образом действует и метод **`Optional.flatMap()`**, если необязательное значение интерпретируется как не имеющее ни одного элемента или же один элемент.

`java.util.Optional` 8

- **`<U> Optional<U> flatMap(Function<? super T,? extends Optional<? extends U>> mapper)`**

  Возвращает результат применения функции ***`mapper()`*** к значению, присутствующему в данном объекте типа **`Optional`**, а иначе — пустой объект типа **`Optional`**.

1.7.7. Преобразование типа `Optional` в поток данных

Метод `stream()` преобразует объект типа `Optional<T>` в поток данных типа `Stream<T>` вообще без элементов или же с единственным элементом. И хотя это вполне возможно, то зачем вообще нужно? Такое преобразование приносит пользу в тех случаях, если методы возвращают результат типа `Optional`. Допустим, имеется поток идентификаторов пользователей и следующий метод их поиска:

```
Optional<User> lookup(String id)
```

Как сформировать поток идентификаторов пользователей, опустив недостоверные идентификаторы? Конечно, для этого можно было бы сначала отсеять недостоверные идентификаторы, а затем применить метод `get()` к оставшимся идентификаторам, как показано ниже.

```
Stream<String> ids = . . .;
Stream<User> users = ids.map(Users::lookup)
                        .filter(Optional::isPresent)
                        .map(Optional::get);
```

Но в данном случае применяются методы `isPresent()` и `get()`, относительно которых ранее были высказаны определенные предостережения. Ниже приведен более изящный способ достичь той же самой цели.

```
Stream<User> users = ids.map(Users::lookup)
                        .flatMap(Optional::stream);
```

Всякий раз, когда метод `stream()` вызывается, он возвращает поток данных вообще без элементов или же с единственным элементом, а метод `flatMap()` все это объединяет в конечный результат. Это означает, что несуществующие пользователи просто опускаются.

НА ЗАМЕТКУ! В этом разделе рассматривается удачный исход, когда метод возвращает значение типа **`Optional`**. Но ведь многие методы ныне возвращают пустое значение **`null`** в отсутствие достоверного результата. Допустим, метод **`Users.classicLookup(id)`** возвращает объект типа **`User`** или пустое значение **`null`**, а не объект типа **`Optional<User>`**. В таком случае пустые значения **`null`** можно, конечно, отсеять, как показано ниже.

```
Stream<User> users = ids.map(Users::classicLookup)
                        .filter(Objects::nonNull);
```

Но вместо этого можно воспользоваться методом **`flatMap()`** следующим образом:

```
Stream<User> users = ids.flatMap(id ->
       Stream.ofNullable(Users.classicLookup(id)));
```

или же таким образом:

```
Stream<User> users = ids.map(Users::classicLookup)
                        .flatMap(Stream::ofNullable);
```

В результате вызова **`Stream.ofNullable(obj)`** возвращается пустой поток данных, если объект **`obj`** оказывается пустым, а иначе — поток, содержащий лишь этот объект.

В примере кода из листинга 1.3 демонстрируется прикладной интерфейс API для необязательного типа `Optional`.

Листинг 1.3. Исходный код из файла **`optional/OptionalTest.java`**

```
1  package optional;
2
3  /**
4   * @version 1.01 2018-05-01
5   * @author Cay Horstmann
6   */
7
8  import java.io.*;
```

```
9  import java.nio.charset.*;
10 import java.nio.file.*;
11 import java.util.*;
12
13 public class OptionalTest
14 {
15   public static void main(String[] args)
16         throws IOException
17   {
18     var contents = new String(Files.readAllBytes(
19       Paths.get("../gutenberg/alice30.txt")),
20                 StandardCharsets.UTF_8);
21     List<String> wordList =
22                  List.of(contents.split("\\PL+"));
23
24     Optional<String> optionalValue = wordList.stream()
25                      .filter(s -> s.contains("fred"))
26                      .findFirst();
27     System.out.println(optionalValue.orElse("No word")
28                      + " contains fred");
29
30     Optional<String> optionalString = Optional.empty();
31     String result = optionalString.orElse("N/A");
32     System.out.println("result: " + result);
33     result = optionalString.orElseGet(() ->
34              Locale.getDefault().getDisplayName());
35     System.out.println("result: " + result);
36     try
37     {
38       result = optionalString.orElseThrow(
39                       IllegalStateException::new);
40       System.out.println("result: " + result);
41     }
42     catch (Throwable t)
43     {
44       t.printStackTrace();
45     }
46
47     optionalValue = wordList.stream()
48                     .filter(s -> s.contains("red"))
49                     .findFirst();
50     optionalValue.ifPresent(s ->
51             System.out.println(s + " contains red"));
52
53     var results = new HashSet<String>();
54     optionalValue.ifPresent(results::add);
55     Optional<Boolean> added =
56              optionalValue.map(results::add);
57     System.out.println(added);
58
59     System.out.println(inverse(4.0)
60               .flatMap(OptionalTest::squareRoot));
61     System.out.println(inverse(-1.0)
62               .flatMap(OptionalTest::squareRoot));
```

```
63      System.out.println(inverse(0.0)
64                .flatMap(OptionalTest::squareRoot));
65      Optional<Double> result2 = Optional.of(-4.0)
66                      .flatMap(OptionalTest::inverse)
67                      .flatMap(OptionalTest::squareRoot);
68      System.out.println(result2);
69    }
70
71    public static Optional<Double> inverse(Double x)
72    {
73      return x == 0 ? Optional.empty()
74                    : Optional.of(1 / x);
75    }
76
77    public static Optional<Double> squareRoot(Double x)
78    {
79      return x < 0 ? Optional.empty()
80                   : Optional.of(Math.sqrt(x));
81    }
82  }
```

java.util.Optional 8

- **<U> Optional<U> flatMap(Function<? super T,Optional<U>> *mapper*)** 9

  Возвращает результат применения функции ***mapper()*** к значению, присутствующему в данном объекте типа **Optional**, или же пустой объект типа **Optional**, если данный объект типа **Optional** оказывается пустым.

1.8. Накопление результатов

По завершении обработки потока данных нередко требуется просмотреть полученные результаты. С этой целью можно вызвать метод `iterate()`, предоставляющий устаревший итератор, которым можно воспользоваться для обхода элементов. С другой стороны, можно вызвать метод `forEach()`, чтобы применить функцию к каждому элементу следующим образом:

```
stream.forEach(System.out::println);
```

В параллельном потоке данных метод `forEach()` выполняет обход элементов в произвольном порядке. Если же их требуется обработать в потоковом порядке, то следует вызвать метод `forEachOrdered()`. Разумеется, в этом случае могут быть утрачены некоторые или даже все преимущества параллелизма.

Но чаще всего результаты требуется накапливать в структуре данных. С этой целью можно вызвать метод `toArray()` и получить элементы из потока данных.

Создать обобщенный массив во время выполнения невозможно, и поэтому в результате вызова `stream.toArray()` возвращается массив типа `Object[]`. Если же требуется массив нужного типа, этому методу следует передать конструктор такого массива:

```
String[] result = stream.toArray(String[]::new);
    // В результате вызова метода stream.toArray()
    // получается массив типа Object[]
```

Для накопления элементов потока данных с другой целью имеется удобный метод `collect()`, принимающий экземпляр класса, реализующего интерфейс `Collector`. *Коллектор* — это объект, накапливающий элементы и производящий результат. В частности, класс `Collectors` предоставляет немало фабричных методов для наиболее употребительных коллекторов. Так, для накопления потока данных в списке или множестве можно воспользоваться коллектором, получаемым в результате вызова `Collectors.toList()`, как показано ниже.

```
List<String> result = stream.collect(Collectors.toList());
```

Аналогично ниже показано, как накопить во множестве элементы из потока данных.

```
Set<String> result = stream.collect(Collectors.toSet());
```

Если же требуется конкретная разновидность получаемого множества, то необходимо сделать следующий вызов:

```
TreeSet<String> result = stream.collect(
        Collectors.toCollection(TreeSet::new));
```

Допустим, требуется накапливать все символьные строки, сцепляя их. С этой целью можно сделать следующий вызов:

```
String result = stream.collect(Collectors.joining());
```

А если требуется разделитель элементов, то его можно передать методу `joining()` следующим образом:

```
String result = stream.collect(Collectors.joining(", "));
```

И если поток данных содержит объекты, отличающиеся от символьных строк, их нужно сначала преобразовать в символьные строки:

```
String result = stream.map(Object::toString)
                      .collect(Collectors.joining(", "));
```

Если результаты обработки потока данных требуется свести к сумме, среднему, максимуму или минимуму, воспользуйтесь методами типа `summarizing(Int|Long|Double)`. Эти методы принимают функцию, преобразующую потоковые объекты в число и возвращающую результат типа `(Int|Long|Double)SummaryStatistics`, одновременно вычисляя сумму, среднее, максимум и минимум, как показано ниже.

```
IntSummaryStatistics summary = stream.collect(
          Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

В примере кода из листинга 1.4 демонстрируется порядок накопления элементов из потока данных.

Листинг 1.4. Исходный код из файла **collecting/CollectingResults.java**

```
1  package collecting;
2
3  /**
4   * @version 1.01 2018-05-01
5   * @author Cay Horstmann
6   */
7
8  import java.io.*;
9  import java.nio.charset.*;
10 import java.nio.file.*;
11 import java.util.*;
12 import java.util.stream.*;
13
14 public class CollectingResults
15 {
16   public static Stream<String> noVowels()
17         throws IOException
18   {
19     var contents = new String(Files.readAllBytes(
20         Paths.get("../gutenberg/alice30.txt")),
21         StandardCharsets.UTF_8);
22     List<String> wordList =
23                 List.of(contents.split("\\PL+"));
24     Stream<String> words = wordList.stream();
25     return words.map(s ->
26                     s.replaceAll("[aeiouAEIOU]", ""));
27   }
28
29   public static <T> void show(String label, Set<T> set)
30   {
31     System.out.print(label + ": "
32                   + set.getClass().getName());
33     System.out.println("[" + set.stream().limit(10)
34               .map(Object::toString)
35               .collect(Collectors.joining(", ")) + "]");
36   }
37
38   public static void main(String[] args)
39                 throws IOException
40   {
41     Iterator<Integer> iter = Stream.iterate(
42              0, n -> n + 1).limit(10).iterator();
43     while (iter.hasNext())
44       System.out.println(iter.next());
45
46     Object[] numbers = Stream.iterate(
47              0, n -> n + 1).limit(10).toArray();
48     System.out.println("Object array:" + numbers);
49       // Обратите внимание, что это массив типа Object[]
50
51     try
52     {
53       var number = (Integer) numbers[0]; // Верно!
```

```
54        System.out.println("number: " + number);
55        System.out.println("The following statement"
56                           + " throws an exception:");
57        // генерируется исключение:
58        var numbers2 = (Integer[]) numbers;
59      }
60      catch (ClassCastException ex)
61      {
62        System.out.println(ex);
63      }
64
65      Integer[] numbers3 = Stream.iterate(0, n -> n + 1)
66                                 .limit(10)
67                                 .toArray(Integer[]::new);
68      System.out.println("Integer array: " + numbers3);
69       // Обратите внимание, что это массив типа Integer[]
70
71      Set<String> noVowelSet =
72                  noVowels().collect(Collectors.toSet());
73      show("noVowelSet", noVowelSet);
74
75      TreeSet<String> noVowelTreeSet = noVowels().collect(
76                   Collectors.toCollection(TreeSet::new));
77      show("noVowelTreeSet", noVowelTreeSet);
78
79      String result = noVowels().limit(10).collect(
80                                 Collectors.joining());
81      System.out.println("Joining: " + result);
82      result = noVowels().limit(10).collect(
83                           Collectors.joining(", "));
84      System.out.println("Joining with commas: " + result);
85
86      IntSummaryStatistics summary = noVowels().collect(
87               Collectors.summarizingInt(String::length));
88      double averageWordLength = summary.getAverage();
89      double maxWordLength = summary.getMax();
90      System.out.println("Average word length: "
91                         + averageWordLength);
92      System.out.println("Max word length: "
93                         + maxWordLength);
94      System.out.println("forEach:");
95      noVowels().limit(10).forEach(System.out::println);
96    }
97  }
```

java.util.stream.BaseStream 8

- **Iterator<T> iterator()**
  Возвращает итератор для получения элементов исходного потока данных. Это оконечная операция.

java.util.stream.Stream 8

- **void forEach(Consumer<? super T> *action*)**

  Вызывает функцию ***action*()** для каждого элемента исходного потока данных. Это оконечная операция.
- **Object[] toArray()**
- **<A> A[] toArray(IntFunction<A[]> *generator*)**

  Возвращают массив объектов или объект типа **A**, если им передается ссылка на конструктор **A[]::new**. Это оконечные операции.
- **<R,A> R collect(Collector<? super T, A, R> *collector*)**

  Накапливает элемент в исходном потоке данных, используя заданный коллектор. Для многих коллекторов в классе **Collectors** имеются фабричные методы.

java.util.stream.Collectors 8

- **static <T> Collector<T, ?, List<T>> toList()**
- **static <T> Collector<T,?,List<T>> toUnmodifiableList()** 10
- **static <T> Collector<T, ?, Set<T>> toSet()**
- **static <T> Collector<T,?,Set<T>> toUnmodifiableSet()** 10

  Возвращают коллекторы, накапливающие элементы в списке или множестве.
- **static <T,C extends Collection<T>> Collector<T, ?, C> toCollection(Supplier<C> *collectionFactory*)**

  Возвращает коллектор, накапливающий элементы в произвольной коллекции. Получает ссылку на конструктор объектов коллекции, например **TreeSet::new**.
- **static Collector<CharSequence, ?, String> joining()**
- **static Collector<CharSequence, ?, String> joining(CharSequence *delimiter*)**
- **static Collector<CharSequence, ?, String> joining(CharSequence *delimiter*, CharSequence *prefix*, CharSequence *suffix*)**

  Возвращают коллектор, соединяющий символьные строки. Заданный разделитель размещается между строками, а префикс и суффикс — перед первой строкой и после последней строки соответственно. Если разделитель, префикс и суффикс не указаны, их места остаются пустыми.
- **static <T> Collector<T, ?, IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> *mapper*)**
- **static<T> Collector<T, ?, LongSummaryStatistics> summarizingLong(ToLongFunction<? superT> *mapper*)**
- **static <T> Collector<T, ?, DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? Super T> *mapper*)**

  Возвращают коллекторы, производящие объект типа **(Int|Long|Double)SummaryStatistics**, из которого получается подсчет, сумма, среднее, максимум и минимум результатов применения функции ***mapper*()** к каждому элементу потока данных.

IntSummaryStatistics 8
LongSummaryStatistics 8
DoubleSummaryStatistics 8

- **long getCount()**

  Возвращает подсчет суммированных элементов.
- **(int|long|double) getSum()**
- **double getAverage()**

  Возвращают сумму или среднее суммированных элементов или же нуль, если элементы отсутствуют.
- **(int|long|double) getMax()**
- **(int|long|double) getMin()**

  Возвращают максимум или минимум суммированных элементов или же значение **Integer|Long|Double).(MAX|MIN)_VALUE**, если элементы отсутствуют.

1.9. Накопление результатов в отображениях

Допустим, имеется поток данных типа `Stream<Person>` и его элементы требуется накапливать в отображении, чтобы в дальнейшем искать людей по их идентификационному номеру. Для этой цели служит метод `Collectors.toMap()`, принимающий в качестве двух своих аргументов функции, чтобы получить ключи и значения из отображения, как показано в следующем примере кода:

```
Map<Integer, String> idToName = people.collect(
    Collectors.toMap(Person::getId, Person::getName));
```

В общем случае, когда значения должны быть конкретными элементами, в качестве второго аргумента данному методу предоставляется функция `Function.identity()` следующим образом:

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(Person::getId, Function.identity()));
```

Если же одному и тому же ключу соответствует больше одного элемента, то возникает конфликт и коллектор генерирует исключение типа `IllegalStateException`. Такое поведение можно изменить, предоставив данному методу в качестве третьего аргумента функцию, разрешающую подобный конфликт и определяющую значение по заданному ключу, исходя из существующего или нового значения. Такая функция может возвратить существующее значение, новое значение или и то и другое.

В приведенном ниже примере создается отображение, содержащее региональные настройки для каждого языка в виде ключа, обозначающего название языка в региональных настройках по умолчанию (например, `"German"`), и значения, обозначающего его локализованное название (например, `"Deutsch"`). В данном примере не учитывается, что один и тот же язык может встретиться дважды (например, немецкий в Германии и Швейцарии), и поэтому в отображении сохраняется лишь первая запись.

```
Stream<Locale> locales =
      Stream.of(Locale.getAvailableLocales());
Map<String, String> languageNames = locales.collect(
    Collectors.toMap(
        Locale::getDisplayLanguage,
        loc -> loc.getDisplayLanguage(loc),
        (existingValue, newValue) -> existingValue));
```

НА ЗАМЕТКУ! В этой главе в качестве структуры данных для хранения региональных настроек употребляется класс **Locale**. Подробнее о региональных настройках речь пойдет в главе 7.

А теперь допустим, что требуется выяснить все языки данной страны. Для этой цели понадобится отображение типа `Map<String, Set<String>>`. Например, значением по ключу `"Switzerland"` является множество `[French, German, Italian]`. Сначала для каждого языка сохраняется одноэлементное множество. А всякий раз, когда обнаруживается новый язык заданной страны, образуется объединение из существующего и нового множеств, как показано ниже.

```
Map<String, Set<String>> countryLanguageSets =
    locales.collect(Collectors.toMap(
        Locale::getDisplayCountry,
        l -> Collections.singleton(l.getDisplayLanguage()),
        (a, b) -> { // объединить множества a и b
        var union = new HashSet<String>(a);
        union.addAll(b);
        return union; }));
```

Более простой способ получения этого отображения будет представлен в следующем разделе. Если же потребуется древовидное отображение типа `TreeMap`, то в качестве четвертого аргумента методу `toMap()` следует предоставить конструктор данного класса. Необходимо также предоставить функцию объединения. Ниже приведен один из примеров из начала этого раздела, переделанный с целью получить отображение типа `TreeMap`.

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(
              Person::getId,
              Function.identity(),
              (existingValue, newValue) ->
                  { throw new IllegalStateException(); },
              TreeMap::new));
```

НА ЗАМЕТКУ! Каждому из вариантов метода **toMap()** соответствует эквивалентный метод **toConcurrentMap()**, получающий параллельное отображение. Единое параллельное отображение применяется в процессе параллельного накопления. Если же общее отображение применяется вместе с параллельным потоком данных, то такой способ оказывается более эффективным, чем объединение множеств. Но в таком случае элементы не накапливаются в потоковом порядке, хотя это обычно не имеет особого значения.

В примере кода из листинга 1.5 демонстрируется накопление потоковых результатов в отображениях.

Листинг 1.5. Исходный код из файла **collecting/CollectingIntoMaps.java**

```
1  package collecting;
2
3  /**
4   * @version 1.00 2016-05-10
5   * @author Cay Horstmann
6   */
7
8  import java.io.*;
9  import java.util.*;
10 import java.util.function.*;
11 import java.util.stream.*;
12
13 public class CollectingIntoMaps
14 {
15   public static class Person
16   {
17     private int id;
18     private String name;
19
20     public Person(int id, String name)
21     {
22       this.id = id;
23       this.name = name;
24     }
25
26     public int getId()
27     {
28       return id;
29     }
30
31     public String getName()
32     {
33       return name;
34     }
35
36     public String toString()
37     {
38       return getClass().getName() + "[id=" + id
39                        + ", name=" + name + "]";
40     }
41   }
42
43   public static Stream<Person> people()
44   {
45     return Stream.of(new Person(1001, "Peter"),
46                      new Person(1002, "Paul"),
47                      new Person(1003, "Mary"));
48   }
49
50   public static void main(String[] args)
51         throws IOException {
52     Map<Integer, String> idToName = people().collect(
53       Collectors.toMap(Person::getId, Person::getName));
54     System.out.println("idToName: " + idToName);
```

```
55
56      Map<Integer, Person> idToPerson = people().collect(
57          Collectors.toMap(Person::getId,
58                           Function.identity()));
59      System.out.println("idToPerson: "
60          + idToPerson.getClass().getName() + idToPerson);
61
62      idToPerson = people().collect(Collectors.toMap(
63          Person::getId, Function.identity(),
64          (existingValue, newValue) ->
65                  {throw new IllegalStateException();},
66          TreeMap::new));
67      System.out.println("idToPerson: "
68                         + idToPerson.getClass().getName()
69                         + idToPerson);
70
71      Stream<Locale> locales =
72            Stream.of(Locale.getAvailableLocales());
73      Map<String, String> languageNames = locales.collect(
74          Collectors.toMap(
75             Locale::getDisplayLanguage,
76             l -> l.getDisplayLanguage(l),
77             (existingValue, newValue) -> existingValue));
78      System.out.println("languageNames: "
79                         + languageNames);
80      locales = Stream.of(Locale.getAvailableLocales());
81      Map<String, Set<String>> countryLanguageSets =
82         locales.collect(Collectors.toMap(
83              Locale::getDisplayCountry,
84              l -> Set.of(l.getDisplayLanguage()),
85              (a, b) ->
86              { // объединить множества a и b
87                   Set<String> union = new HashSet<>(a);
88                   union.addAll(b);
89                   return union;
90               }));
91      System.out.println("countryLanguageSets: "
92                         + countryLanguageSets);
93   }
94 }
```

java.util.stream.Collectors 8

- **static<T,K,U> Collector<T, ?, Map<K, U>> toMap(Function<? superT, ? extendsK> *keyMapper*, Function<? super T, ? extends U> *valueMapper*)**
- **static<T, K, U> Collector<T, ?, Map<K, U>> toMap(Function<? superT, ? extendsK> *keyMapper*, Function<? super T, ? extends U> *valueMapper*, BinaryOperator<U> *mergeFunction*)**
- **static <T, K, U, M extends Map<K, U>> Collector<T, ?, M> toMap(Function<? super T, ? extends K> *keyMapper*, Function<? super T, ? extends U> *valueMapper*, BinaryOperator<U> *mergeFunction*, Supplier<M> *mapSupplier*)**

java.util.stream.Collectors 8 *(окончание)*

- **static <T,K,U> Collector<T,?,Map<K,U>> toUnmodifiableMap(Function<? super T,? extends K> *keyMapper*, Function<? super T,? extends U> *valueMapper*)** 10
- **static <T,K,U> Collector<T,?,Map<K,U>> toUnmodifiableMap(Function<? super T,? extends K> *keyMapper*, Function<? super T,? extends U> *valueMapper*, BinaryOperator<U> *mergeFunction*)** 10
- **static <T, K, U> Collector<T,?,ConcurrentMap<K, U>> toConcurrentMap(Function<? super T, ? Extends K> *keyMapper*, Function<? super T, ? extends U> *valueMapper*)**
- **static <T, K, U> Collector<T, ?, ConcurrentMap<K, U>> toConcurrentMap(Function<? super T, ? Extends K> *keyMapper*, Function<? super T, ? extends U> *valueMapper*, BinaryOperator<U> *mergeFunction*)**
- **static <T, K, U, M extends ConcurrentMap<K,U>> Collector<T, ?, M> toConcurrentMap(Function<? super T, ? extends K> *keyMapper*, Function<? super T, ? extends U> *valueMapper*, BinaryOperator<U> *mergeFunction*, Supplier<M> *mapSupplier*)**

  Возвращают коллектор, производящий обычное, неизменяемое или параллельное отображение. Функции ***keyMapper()*** и ***valueMapper()*** применяются к каждому накапливаемому элементу, возвращая запись в виде пары "ключ–значение" из результирующего отображения. По умолчанию генерируется исключение типа **IllegalStateException**, когда два элемента порождают одинаковый ключ. Вместо этого можно применить функцию ***mergeFunction()***, объединяющую значения по одному и тому же ключу. По умолчанию получается результирующее отображение типа **HashMap** или **ConcurrentHashMap**. Но вместо этого можно предоставить функцию ***mapSupplier()***, возвращающую требующийся экземпляр отображения.

1.10. Группирование и разделение

В предыдущем разделе было показано, как накапливаются все языки заданной страны. Но этот процесс оказался несколько трудоемким, поскольку для каждого значения из отображения пришлось сначала сформировать одноэлементное множество, а затем указать порядок объединения существующего и нового значений. Нередко из значений с одинаковыми характеристиками образуются группы, и этот процесс непосредственно поддерживается методом `groupingBy()`.

Рассмотрим задачу группирования региональных настроек по странам. Сначала образуется следующее отображение:

```
Map<String, List<Locale>> countryToLocales =
    locales.collect(Collectors.groupingBy(
                    Locale::getCountry));
```

Функция `Locale::getCountry()` исполняет роль *классификатора* группирования. Затем все региональные настройки можно отыскать по заданному коду страны, как показано в следующем примере кода:

```
List<Locale> swissLocales = countryToLocales.get("CH");
   // получить региональные настройки [it_CH, de_CH, fr_CH]
```

НА ЗАМЕТКУ! Как известно, все региональные настройки состоят из кода языка (например, код **en** обозначает английский язык) и кода страницы (например, код **US** обозначает Соединенные Штаты). Так, региональные настройки **en_US** описывают английский язык в Соединенных Штатах, а региональные настройки **en_IE** — английский язык в Ирландии. Некоторым странам требуется несколько региональных настроек. Например, региональные настройки **ga_IE**, описывающие гэльский язык в Ирландии в дополнение к упомянутым выше региональным настройкам **en_IE**. А для Швейцарии требуются три региональные настройки, как было показано в предыдущем разделе.

Когда функция классификатора оказывается предикатной (т.е. функцией, возвращающей логическое значение типа `boolean`), элементы потока данных разделяются на основной список с элементами, для которых функция возвращает логическое значение `true`, и дополнительный список. В данном случае эффективнее воспользоваться методом `partitioningBy()`, чем методом `groupingBy()`. Так, в следующем примере кода все региональные настройки разделяются на те, которые описывают английский язык, и все остальные:

```
Map<Boolean, List<Locale>> englishAndOtherLocales =
    locales.collect(Collectors.partitioningBy(
                    l -> l.getLanguage().equals("en")));
List<Locale>> englishLocales =
              englishAndOtherLocales.get(true);
```

НА ЗАМЕТКУ! Если вызвать метод **groupingByConcurrent()**, то в конечном итоге будет получено отображение, которое заполняется параллельно, если оно применяется вместе с параллельным потоком данных. В этом отношении данный метод очень похож на метод **toConcurrentMap()**.

java.util.stream.Collectors 8

- **static<T,K> Collector<T, ?, Map<K, List<T>>> groupingBy(Function<? superT, ? extendsK> *classifier*)**
- **static <T,K> Collector<T, ?, ConcurrentMap<K, List<T>>> groupingByConcurrent(Function<? super T, ? extends K> *classifier*)**

  Возвращают коллектор, производящий обычное или параллельное отображение, где ключи являются результатом применения функции ***classifier*()** ко всем накапливаемым элементам, а значения — списками элементов с одинаковым ключом.
- **static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> *predicate*)**

  Возвращает коллектор, производящий отображение, где ключи принимают логическое значение **true**/**false**, а значения являются списками элементов, совпадающих или не совпадающих с заданным предикатом.

1.11. Нисходящие коллекторы

Метод `groupingBy()` формирует множество, значениями которого являются списки. Если требуется обработать эти списки каким-то образом, то следует предоставить *нисходящий коллектор*. Так, если вместо списков требуются множества, можно воспользоваться коллектором `Collectors.toSet()` следующим образом:

```
Map<String, Set<Locale>> countryToLocaleSet =
    locales.collect(groupingBy(
                    Locale::getCountry, toSet()));
```

НА ЗАМЕТКУ! В данном и последующих примерах из этого раздела предполагается статический импорт `java.util.stream.Collectors.*`, чтобы упростить выражения и сделать их более удобочитаемыми.

Для сведения сгруппированных элементов к числам предоставляется ряд следующих коллекторов:

- **`counting()`** — производит подсчет накопленных элементов. Так, в следующем примере кода подсчитывается количество региональных настроек для каждой страны:

```
Map<String, Long> countryToLocaleCounts = locales.collect(
    groupingBy(Locale::getCountry, counting()));
```

- **`summing(Int|Long|Double)`** — принимает в качестве аргумента функцию, применяет ее к элементам нисходящего потока данных и получает их сумму. Так, в следующем примере кода вычисляется суммарное население каждого штата из потока городов:

```
Map<String, Integer> stateToCityPopulation =
    cities.collect(groupingBy(City::getState,
                   summingInt(City::getPopulation)));
```

- **`maxBy()` и `minBy()`** — принимают в качестве аргумента компаратор и получают максимальный и минимальный элементы из нисходящего потока данных. Так, в следующем примере кода получается самый крупный город в каждом штате:

```
Map<String, City> stateToLargestCity = cities.collect(
    groupingBy(City::getState,
        maxBy(Comparator.comparing(City::getPopulation))));
```

- **`collectingAndThen()`** — вводит завершающую стадию накопления. Так, если требуется выяснить, сколько имеется отдельных результатов, их можно накопить сначала в множестве, а затем вычислить его размер, как показано ниже.

```
Map<Character, Integer> stringCountsByStartingLetter =
   strings.collect(groupingBy(s -> s.charAt(0),
                   collectingAndThen(toSet(), Set::size)));
```

Совсем иначе действует коллектор, реализуемый методом `mapping()`. Он применяет функцию к каждому накапливаемому элементу и передает полученные результаты нисходящему коллектору, как демонстрируется в приведенном ниже примере кода.

```
Map<Character, Set<Integer>> stringLengthsByStartingLetter
    = strings.collect(groupingBy(s -> s.charAt(0),
                      mapping(String::length, toSet())));
```

В данном примере строки группируются по их первому символу. Для каждой группы определяется ее длина, которая накапливается в множестве.

Метод `mapping()` позволяет также решить изящнее задачу из предыдущего раздела — собрать все языки, употребляемые в стране. В предыдущем разделе вместо метода `groupingBy()` применялся метод `toMap()`. А в приведенном ниже решении отпадает необходимость объединять отдельные множества.

```
Map<String, Set<String>> countryToLanguages =
    locales.collect(groupingBy(Locale::getDisplayCountry,
            mapping(Locale::getDisplayLanguage, toSet())));
```

Для применения вместе с функциями, возвращающими потоки данных, имеется также коллектор, реализуемый методом `flatMapping()`.

Если функция группирования или отображения возвращает значение типа `int`, `long` или `double`, элементы можно накопить в объекте суммарной статистики, как пояснялось в разделе 1.8. Ниже показано, как это делается. А затем из объектов суммарной статистики каждой группы можно получить суммарное, подсчитанное, среднее, минимальное и максимальное значения функции.

```
Map<String, IntSummaryStatistics>
    stateToCityPopulationSummary = cities.collect(
        groupingBy(City::getState,
                   summarizingInt(City::getPopulation)));
```

Коллектор, реализуемый методом `filtering()`, применяет фильтр к каждой группе, как демонстрируется в следующем примере кода:

```
Map<String, Set<City>> largeCitiesByState = cities.collect(
    groupingBy(City::getState,
               filtering(c -> c.getPopulation() > 500000,
                         toSet()))); // штаты без крупных
                    // городов, но с пустыми множествами
```

НА ЗАМЕТКУ! Имеются три варианта метода **`reducing()`**, выполняющие общие операции сведения, описываемые далее в разделе 1.12.

Коллекторы можно эффективно сочетать вместе, но в итоге получаются весьма запутанные выражения. Поэтому их лучше всего использовать вместе с методом `groupingBy()` или `partitioningBy()` для обработки значений, преобразуемых из нисходящего потока данных. В противном случае непосредственно в потоках данных применяются такие методы, как `map()`, `reduce()`, `count()`, `max()` или `min()`.

Применение нисходящих коллекторов демонстрируется в примере кода из листинга 1.6.

Листинг 1.6. Исходный код из файла **`collecting/DownstreamCollectors.java`**

```
1  package collecting;
2
3  /**
4   * @version 1.00 2016-05-10
5   * @author Cay Horstmann
6   */
7
8  import static java.util.stream.Collectors.*;
```

```
9
10  import java.io.*;
11  import java.nio.file.*;
12  import java.util.*;
13  import java.util.stream.*;
14
15  public class DownstreamCollectors
16  {
17
18    public static class City
19    {
20       private String name;
21       private String state;
22       private int population;
23
24       public City(String name, String state,
25                   int population)
26       {
27         this.name = name;
28         this.state = state;
29         this.population = population;
30       }
31
32      public String getName()
33      {
34        return name;
35      }
36
37      public String getState()
38      {
39        return state;
40      }
41
42      public int getPopulation()
43      {
44        return population;
45      }
46    }
47
48    public static Stream<City> readCities(String filename)
49           throws IOException
50    {
51      return Files.lines(Paths.get(filename))
52                  .map(l -> l.split(", "))
53                  .map(a -> new City(a[0], a[1],
54                       Integer.parseInt(a[2])));
55    }
56
57    public static void main(String[] args)
58           throws IOException
59    {
60      Stream<Locale> locales =
61             Stream.of(Locale.getAvailableLocales());
62      locales = Stream.of(Locale.getAvailableLocales());
63      Map<String, Set<Locale>> countryToLocaleSet =
64          locales.collect(groupingBy(
65                          Locale::getCountry, toSet()));
```

```
66      System.out.println("countryToLocaleSet: "
67                         + countryToLocaleSet);
68
69      locales = Stream.of(Locale.getAvailableLocales());
70      Map<String, Long> countryToLocaleCounts =
71          locales.collect(groupingBy(
72                           Locale::getCountry, counting()));
73      System.out.println("countryToLocaleCounts: "
74                         + countryToLocaleCounts);
75
76      Stream<City> cities = readCities("cities.txt");
77      Map<String, Integer> stateToCityPopulation =
78          cities.collect(groupingBy(City::getState,
79                         summingInt(City::getPopulation)));
80      System.out.println("stateToCityPopulation: "
81                         + stateToCityPopulation);
82
83      cities = readCities("cities.txt");
84      Map<String, Optional<String>>
85          stateToLongestCityName = cities.collect(
86        groupingBy(City::getState,
87          mapping(City::getName,
88          maxBy(Comparator.comparing(String::length)))));
89      System.out.println("stateToLongestCityName: "
90                         + stateToLongestCityName);
91
92      locales = Stream.of(Locale.getAvailableLocales());
93      Map<String, Set<String>> countryToLanguages =
94        locales.collect(groupingBy(
95          Locale::getDisplayCountry,
96          mapping(Locale::getDisplayLanguage, toSet())));
97      System.out.println("countryToLanguages: "
98                         + countryToLanguages);
99
100     cities = readCities("cities.txt");
101     Map<String, IntSummaryStatistics>
102        stateToCityPopulationSummary =
103           cities.collect(groupingBy(City::getState,
104                  summarizingInt(City::getPopulation)));
105     System.out.println(
106                stateToCityPopulationSummary.get("NY"));
107
108     cities = readCities("cities.txt");
109     Map<String, String> stateToCityNames =
110         cities.collect(groupingBy(City::getState,
111                reducing("", City::getName, (s, t) ->
112                  s.length() == 0 ? t : s + ", " + t)));
113
114     cities = readCities("cities.txt");
115     stateToCityNames = cities.collect(groupingBy(
116                City::getState,
117                mapping(City::getName, joining(", "))));
118     System.out.println("stateToCityNames: "
119                        + stateToCityNames);
120   }
121 }
```

`java.util.stream.Collectors` 8

- **`public static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`**

  Возвращает коллектор, производящий отображение. Ключи получаются в результате применения функции ***`classifier()`*** ко всем накапливаемым элементам, а значения — в результате накапливания элементов по одному и тому же ключу с помощью нисходящего коллектора.
- **`static <T> Collector<T,?,Long> counting()`**

  Возвращает коллектор, подсчитывающий накапливаемые элементы.
- **`static <T> Collector<T, ?, Integer> summingInt(ToIntFunction<? super T> mapper)`**
- **`static <T> Collector<T, ?, Long> summingLong(ToLongFunction<? super T> mapper)`**
- **`static <T> Collector<T, ?, Double> summingDouble(ToDoubleFunction<? super T> mapper)`**

  Возвращают коллектор, вычисляющий сумму значений, получаемых в результате применения функции ***`mapper()`*** к накапливаемым элементам.
- **`static <T> Collector<T, ?, Optional<T>> maxBy(Comparator<? super T> comparator)`**
- **`static <T> Collector<T, ?, Optional<T>> minBy(Comparator<? super T> comparator)`**

  Возвращают коллектор, вычисляющий максимальный или минимальный из накапливаемых элементов, используя порядок расположения, который задает ***`comparator`***.
- **`static <T, A, R, RR> Collector<T, A, RR> collectingAndThen(Collector<T, A, RR> downstream, Function<R, RR> finisher)`**

  Возвращает коллектор, сначала направляющий элементы в коллектор, а затем применяющий функцию ***`finisher()`*** к полученному результату.
- **`static <T, U, A, R> Collector<T, ?, R> mapping(Function<? super T, ? extends U> mapper, Collector<? super U, A, R> downstream)`**

  Возвращает коллектор, вызывающий функцию ***`mapper()`*** для каждого элемента и передающий полученные результаты нисходящему коллектору.
- **`static <T,U,A,R> Collector<T,?,R> flatMapping(Function<? super T,? extends Stream<? extends U>> mapper, Collector<? super U,A,R> downstream)`**

  Возвращает коллектор, вызывающий функцию ***`mapper()`*** для каждого элемента и передающий полученные результаты нисходящему коллектору.
- **`static <T,A,R> Collector<T,?,R> filtering(Predicate<? super T> predicate, Collector<? super T,A,R> downstream)`**

  Возвращает коллектор, передающий нисходящему коллектору элементы, получаемые в результате применения функции ***`predicate()`***.

1.12. Операции сведения

Метод `reduce()` реализует общий механизм для вычисления значения из потока данных. В простейшей форме он принимает двоичную функцию

и применяет ее, начиная с первых двух элементов потока данных. Этот механизм проще всего пояснить на следующем примере функции суммирования:

```
List<Integer> values = ...;
Optional<Integer> sum = values.stream()
                              .reduce((x, y) -> x + y);
```

В данном примере метод `reduce()` вычисляет сумму $v_0 + v_1 + v_2 + ...$, где v_i — элементы потока данных. Этот метод возвращает объект типа `Optional`, поскольку достоверный результат недостижим, если поток данных пуст.

НА ЗАМЕТКУ! В данном примере можно сделать вызов **`reduce(Integer::sum)`** вместо вызова **`reduce((x, y) -> x + y)`**.

В более общем смысле любую операцию, объединяющую частичный результат `x` со следующим значением `y`, можно использовать для получения нового частичного результата. Операции сведения можно рассматривать и под иным углом зрения. Так, если имеется операция сведения *op*, то она дает результат v_0 *op* v_1 *op* v_2 *op*..., где v_i *op* v_{i+1} обозначает вызов функции $op(v_i, v_{i+1})$. Практическую пользу могут принести многие операции сведения, в том числе сложение, умножение, сцепление символьных строк, получение максимума и минимума, объединение и пересечение множеств.

Если операцию сведения требуется выполнить над параллельными потоками данных, такая операция должна быть *ассоциативной*. Это означает, что порядок объединения элементов в такой операции не имеет никакого значения. В математическом обозначении операция $(x\ op\ y)\ op\ z$ должна быть равнозначна операции $x\ op\ (y\ op\ z)$. Примером операции, которая не является ассоциативной, служит вычитание. Так, `(6 - 3) - 2` ≠ `6 - (3 - 2)`.

Нередко имеется *тождественный элемент e* вроде $e\ op\ x = x$, и он может быть использован в качестве отправной точки для вычисления. Например, `0` является тождественным элементом операции сложения. Ниже приведена вторая форма вызова метода `reduce()`. Тождественный элемент возвращается в том случае, если поток данных пуст и больше не нужно обращаться к классу `Optional`.

```
List<Integer> values = ...;
Integer sum = values.stream().reduce(0, (x, y) -> x + y)
    // Вычисляет результат 0 + v0 + v1 + v2 + ...
```

А теперь допустим, что имеется поток объектов и требуется получить сумму некоторых свойств, например, длину всех символьных строк в потоке. Для этой цели не годится простая форма метода `reduce()`, поскольку в ней требуется функция `(T, T) -> T` с одинаковыми типами аргументов и возвращаемого результата. Но в данном случае имеются два разных типа: `String` — для элементов потока данных и `int` — для накапливаемого результата. На этот случай имеется отдельная форма вызова метода `reduce()`.

Прежде всего нужно предоставить функцию накопления `(total, word) -> total + word.length()`, которая вызывается повторно, образуя сумму нарастающим итогом. Но если вычисление этой суммы распараллелено, то оно разделяется на несколько параллельных вычислений, результаты которых должны быть

объединены. Для этой цели предоставляется вторая функция. Ниже приведена полная форма вызова метода `reduce()` в данном случае.

```
int result = words.reduce(0,
             (total, word) -> total + word.length(),
             (total1, total2) -> total1 + total2);
```

НА ЗАМЕТКУ! На практике методом **reduce()** приходится пользоваться нечасто. Ведь намного проще преобразовать исходный поток символьных строк в поток чисел и воспользоваться одним из методов для вычисления суммы, максимума или минимума. (Подробнее потоки чисел рассматриваются далее в разделе 1.13.) В данном конкретном случае можно было бы сделать вызов **words.mapToInt(String::length).sum()**. Это было бы проще и эффективнее, поскольку не потребовало бы упаковки.

НА ЗАМЕТКУ! Иногда метод **reduce()** оказывается недостаточно обобщенным. Допустим, требуется накопить результаты во множестве типа **BitSet**. Если распараллелить эту коллекцию, то разместить ее элементы в одном множестве типа **BitSet** не удастся, поскольку объект типа **BitSet** не является потокобезопасным. Именно поэтому нельзя воспользоваться методом **reduce()**. Каждый сегмент исходной коллекции должен начинаться со своего пустого множества, а методу **reduce()** можно предоставить только одно тождественное значение. В таком случае следует воспользоваться методом **collect()**, который принимает следующие аргументы.

Поставщик для получения новых экземпляров целевого объекта. Например, конструктор для построения хеш-множества.

Накопитель, вводящий элемент в целевой объект. Например, метод **add()**.

Объединитель, соединяющий два объекта в один. Например, метод **addAll()**.

Ниже показано, каким образом метод **collect()** вызывается для множества битов.

```
BitSet result = stream.collect(BitSet::new, BitSet::set,
                               BitSet::or);
```

java.util.Stream 8

- **Optional<T> reduce(BinaryOperator<T>** ***accumulator*****)**
- **T reduce(T** ***identity*****, BinaryOperator<T>** ***accumulator*****)**
- **<U> U reduce(U** ***identity*****, BiFunction<U, ? super T, U>** ***accumulator*****, BinaryOperator<U>** ***combiner*****)**

  Формируют накапливаемый итог элементов потока данных с помощью заданной функции ***accumulator()***. Если же предоставляется аргумент ***identity***, то он становится первым накапливаемым значением. А если в качестве аргумента предоставляется функция ***combiner()***, она может быть использована для объединения итогов по сегментам потока данных, которые накапливаются отдельно.

- **<R> Rcollect(Supplier<R>** ***supplier*****, BiConsumer<R, ? superT>** ***accumulator*****, BiConsumer<R, R>** ***combiner*****)**

  Накапливает элементы в результат типа **R**. Для каждого сегмента потока данных вызывается функция ***supplier()***, предоставляющая первоначальный результат. Функция ***accumulator()*** вызывается для добавления к нему элементов изменчивым способом, а функция ***combiner()*** — для объединения обоих результатов.

1.13. Потоки данных примитивных типов

До сих пор целочисленные значения накапливались в потоке данных типа `Stream<Integer>`, несмотря на то, что заключать каждое целочисленное значение в объект-оболочку совершенно неэффективно. Это же относится и к другим примитивным типам данных `double`, `float`, `long`, `short`, `char`, `byte` и `boolean`. В библиотеке потоков данных имеются специализированные классы `IntStream`, `LongStream` и `DoubleStream`, позволяющие сохранять значения примитивных типов непосредственно, не прибегая к помощи оболочек. Так, если требуется сохранить значения типа `short`, `char`, `byte` и `boolean`, достаточно воспользоваться классом `IntStream`, а для хранения значений типа `float` — классом `DoubleStream`.

Чтобы создать поток данных типа `IntStream`, достаточно вызвать методы `IntStream.of()` и `Arrays.stream()` следующим образом:

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
stream = Arrays.stream(values, from, to);
    // массив values относится к типу int[]
```

К потокам данных примитивных типов, как и к потокам объектов, можно применять статические методы `generate()` и `iterate()`. Кроме того, в классах `IntStream` и `LongStream` имеются статические методы `range()` и `rangeClosed()`, генерирующие диапазоны целочисленных значений с единичным шагом, как показано ниже.

```
IntStream zeroToNinetyNine = IntStream.range(0, 100);
    // Верхний предел исключительно
IntStream zeroToHundred = IntStream.rangeClosed(0, 100);
    // Верхний предел включительно
```

В интерфейсе `CharSequence` имеются методы `codePoints()` и `chars()`, получающие поток типа `IntStream` кодов символов в Юникоде или кодовых единиц в кодировке UTF-16. (Подробнее о кодировках символов — в главе 2.) Ниже приведен пример применения метода `codePoints()`.

```
String sentence = "\uD835\uDD46 is the set of octonions.";
    // \uD835\uDD46 — это кодировка UTF-16 знака 𝕆,
    // обозначающего октонионы в Юникоде (U+1D546)

IntStream codes = sentence.codePoints();
    // Поток шестнадцатеричных значений
    // 1D546 20 69 73 20 . . .
```

Поток объектов можно преобразовать в поток данных примитивных типов с помощью методов `mapToInt()`, `mapToLong()` или `mapToDouble()`. Так, если имеется поток символьных строк и их длины требуется обработать как целочисленные значения, это можно сделать и средствами класса `IntStream` следующим образом:

```
Stream<String> words = ...;
IntStream lengths = words.mapToInt(String::length);
```

Чтобы преобразовать поток данных примитивного типа в поток объектов, достаточно воспользоваться методом `boxed()` следующим образом:

```
Stream<Integer> integers = IntStream.range(0, 100).boxed();
```

Как правило, методы для потоков данных примитивных типов аналогичны методам для потоков объектов. Ниже перечислены наиболее существенные их отличия.

- Методы типа `toArray` возвращают массивы примитивных типов.
- Методы, возвращающие результат необязательного типа, возвращают значение типа `OptionalInt`, `OptionalLong` или `OptionalDouble`. Классы этих типов аналогичны классу `Optional`, но у них имеются методы `getAsInt()`, `getAsLong()` и `getAsDouble()` вместо метода `get()`.
- Имеются методы `sum()`, `average()`, `max()` и `min()`, возвращающие сумму, среднее, максимум и минимум соответственно. Эти методы не определены для потоков объектов.
- Метод `summaryStatistics()` возвращает объект типа `IntSummaryStatistics`, `LongSummaryStatistics` или `DoubleSummaryStatistics`, способный одновременно сообщать о сумме, среднем, максимуме и минимуме в потоке данных.

НА ЗАМЕТКУ! В классе **`Random`** имеются методы **`ints()`**, **`longs()`** и **`doubles()`**, возвращающие потоки данных примитивных типов, состоящие из случайных чисел. Если же случайные числа потребуются в параллельных потоках данных, в таком случае следует воспользоваться классом **`SplittableRandom`**.

В примере кода из листинга 1.7 демонстрируется применение элементов прикладного программного интерфейса API для потоков данных примитивных типов.

Листинг 1.7. Исходный код из файла **`streams/PrimitiveTypeStreams.java`**

```
1  package streams;
2
3  /**
4   * @version 1.01 2018-05-01
5   * @author Cay Horstmann
6   */
7
8  import java.io.IOException;
9  import java.nio.charset.StandardCharsets;
10 import java.nio.file.Files;
11 import java.nio.file.Path;
12 import java.nio.file.Paths;
13 import java.util.stream.Collectors;
14 import java.util.stream.IntStream;
15 import java.util.stream.Stream;
16
17 public class PrimitiveTypeStreams
18 {
19   public static void show(String title, IntStream stream)
20   {
21     final int SIZE = 10;
22     int[] firstElements =
23           stream.limit(SIZE + 1).toArray();
24     System.out.print(title + ": ");
```

```
25      for (int i = 0; i < firstElements.length; i++)
26      {
27        if (i > 0) System.out.print(", ");
28        if (i < SIZE) System.out.print(firstElements[i]);
29        else System.out.print("...");
30      }
31      System.out.println();
32    }
33
34    public static void main(String[] args)
35          throws IOException
36    {
37      IntStream is1 = IntStream.generate(() ->
38                      (int) (Math.random() * 100));
39      show("is1", is1);
40      IntStream is2 = IntStream.range(5, 10);
41      show("is2", is2);
42      IntStream is3 = IntStream.rangeClosed(5, 10);
43      show("is3", is3);
44
45      Path path = Paths.get("../gutenberg/alice30.txt");
46      var contents = new String(Files.readAllBytes(path),
47                                StandardCharsets.UTF_8);
48
49      Stream<String> words =
50                    Stream.of(contents.split("\\PL+"));
51      IntStream is4 = words.mapToInt(String::length);
52      show("is4", is4);
53      var sentence = "\uD835\uDD46 is the set "
54                   + "of octonions.";
55      System.out.println(sentence);
56      IntStream codes = sentence.codePoints();
57      System.out.println(codes.mapToObj(c ->
58                         String.format("%X ", c)).collect(
59                                 Collectors.joining()));
60
61      Stream<Integer> integers =
62            IntStream.range(0, 100).boxed();
63      IntStream is5 =
64            integers.mapToInt(Integer::intValue);
65      show("is5", is5);
66    }
67  }
```

java.util.stream.IntStream 8

- **static IntStream range(int *startInclusive*, int *endExclusive*)**
- **static IntStream rangeClosed(int *startInclusive*, int *endInclusive*)**

  Возвращают поток данных типа **IntStream** с целочисленными элементами в заданном диапазоне.
- **static IntStream of(int... *values*)**

  Возвращает поток данных типа **IntStream** с заданными элементами.

java.util.stream.IntStream 8 *(окончание)*

- **int[] toArray()**

  Возвращает массив, состоящий из элементов исходного потока данных.
- **int sum()**
- **OptionalDouble average()**
- **OptionalInt max()**
- **OptionalInt min()**
- **IntSummaryStatistics summaryStatistics()**

  Возвращают сумму, среднее, максимум, минимуму элементов исходного потока данных или объект, из которого могут быть получены эти четыре результата.
- **Stream<Integer> boxed()**

  Возвращает поток объектов-оболочек для элементов исходного потока данных.

java.util.stream.LongStream 8

- **static LongStream range(long *startInclusive*, long *endExclusive*)**
- **static LongStream rangeClosed(long *startInclusive*, long *endInclusive*)**

  Возвращают поток данных типа **LongStream** с целочисленными элементами в заданном диапазоне.
- **static LongStream of(long... *values*)**

  Возвращает поток данных типа **LongStream** с заданными элементами.
- **long[] toArray()**

  Возвращает массив, состоящий из элементов исходного потока данных.
- **long sum()**
- **OptionalDouble average()**
- **OptionalLong max()**
- **OptionalLong min()**
- **LongSummaryStatistics summaryStatistics()**

  Возвращают сумму, среднее, максимум, минимуму элементов исходного потока данных или объект, из которого могут быть получены эти четыре результата.
- **Stream<Long> boxed()**

  Возвращает поток объектов-оболочек для элементов исходного потока данных.

java.util.stream.DoubleStream 8

- **static DoubleStream of(double... *values*)**

  Возвращает поток данных типа **DoubleStream** с заданными элементами.

java.util.stream.DoubleStream 8 *(окончание)*

- **double[] toArray()**

  Возвращает массив, состоящий из элементов исходного потока данных.
- **double sum()**
- **OptionalDouble average()**
- **OptionalDouble max()**
- **OptionalDouble min()**
- **DoubleSummaryStatistics summaryStatistics()**

  Возвращают сумму, среднее, максимум, минимуму элементов исходного потока данных или объект, из которого могут быть получены эти четыре результата.
- **Stream<Double> boxed()**

  Возвращает поток объектов-оболочек для элементов исходного потока данных.

java.lang.CharSequence 1.0

- **IntStream codePoints()** 8

  Возвращает поток всех кодовых точек исходной символьной строки в Юникоде.

java.util.Random 1.0

- **IntStream ints()**
- **IntStream ints(int *randomNumberOrigin*, int *randomNumberBound*)** 8
- **IntStream ints(long *streamSize*)** 8
- **IntStream ints(long *streamSize*, int *randomNumberOrigin*, int *randomNumberBound*)** 8
- **LongStream longs()** 8
- **LongStream longs(long *randomNumberOrigin*, long *randomNumberBound*)** 8
- **LongStream longs(long *streamSize*)** 8
- **LongStream longs(long *streamSize*, long *randomNumberOrigin*, long *randomNumberBound*)** 8
- **DoubleStream doubles()** 8
- **DoubleStream doubles(double *randomNumberOrigin*, double *randomNumberBound*)** 8
- **DoubleStream doubles(long *streamSize*)** 8
- **DoubleStream doubles(long *streamSize*, double *randomNumberOrigin*, double *randomNumberBound*)** 8

  Возвращают потоки произвольных чисел. Если указан аргумент ***streamSize***, возвращается конечный поток с заданным количеством элементов. Если же предоставлены границы, то возвращается поток с элементами в пределах от ***randomNumberOrigin*** (включительно) до ***randomNumberBound*** (исключительно).

java.util.Optional(Int|Long|Double) 8

- **static Optional(Int|Long|Double) of((int|long|double) *value*)**
  Возвращает необязательный объект с предоставленным значением указанного примитивного типа.
- **(int|long|double) getAs(Int|Long|Double)()**
  Возвращает значение данного необязательного объекта или генерирует исключение типа **NoSuchElementException**, если этот объект оказывается пустым.
- **(int|long|double) orElse((int|long|double) *other*)**
- **(int|long|double) orElseGet((Int|Long|Double)Supplier *other*)**
  Возвращают значение данного необязательного объекта или альтернативное значение, если этот объект оказывается пустым.
- **void ifPresent((Int|Long|Double)Consumer *consumer*)**
  Передает значение данного необязательного объекта функции ***consumer*()**, если этот объект оказывается непустым.

java.util.(Int|Long|Double)SummaryStatistics 8

- **long getCount()**
- **(int|long|double) getSum()**
- **double getAverage()**
- **(int|long|double) getMax()**
- **(int|long|double) getMin()**
  Возвращают подсчет, сумму, среднее, максимум и минимум накапливаемых элементов исходного потока данных.

1.14. Параллельные потоки данных

Потоки данных упрощают распараллеливание групповых операций. Этот процесс происходит в основном автоматически, но требует соблюдения немногих правил. Прежде всего, нужно иметь в своем распоряжении параллельный поток данных. Получить параллельный поток данных можно из любой коллекции с помощью метода `Collection.parallelStream()` следующим образом:

```
Stream<String> parallelWords = words.parallelStream();
```

Более того, метод `parallel()` преобразует любой последовательный поток данных в параллельный поток, как показано ниже. При выполнении оконечного метода поток данных действует в параллельном режиме, и поэтому промежуточные операции в этом потоке распараллеливаются.

```
Stream<String> parallelWords =
               Stream.of(wordArray).parallel();
```

Когда потоковые операции выполняются параллельно, цель состоит в том, чтобы получить в итоге такой же результат, как и в том случае, если бы они

выполнялись последовательно. Очень важно, чтобы эти операции можно было выполнять в произвольном порядке.

Допустим, требуется подсчитать все короткие слова в потоке символьных строк. В приведенном ниже примере демонстрируется, как *не* следует решать эту задачу.

```
int[] shortWords = new int[12];
words.parallelStream().forEach(s ->
      { if (s.length() < 12) shortWords[s.length()]++; });
        // ОШИБКА: состояние гонок!
System.out.println(Arrays.toString(shortWords));
```

Приведенный выше код написан очень скверно. Функция, передаваемая методу `forEach()`, выполняется параллельно в нескольких потоках исполнения, в каждом из которых обновляется разделяемый ими общий массив. Как будет показано в главе 12, это классическое *состояние гонок*. Если выполнить данный код многократно, то в результате каждого его выполнения, вероятнее всего, будет получена совсем другая последовательность подсчитанных коротких слов, причем каждый раз неверная.

В обязанности программиста входит обеспечение надежного выполнения в параллельном режиме функций, передаваемых для распараллеливания операций в потоке данных. Для этого лучше всего избегать изменяемого состояния. В следующем примере кода наглядно показано, что вычисления можно надежно распараллелить, если сгруппировать символьные строки по длине и подсчитать их:

```
Map<Integer, Long> shortWordCounts =
    words.parallelStream()
         .filter(s -> s.length() < 10)
         .collect(groupingBy(String::length, counting()));
```

По умолчанию потоки данных, получаемые из упорядоченных коллекций (массивов и списков), диапазонов, генераторов, итераторов или в результате вызова метода `Stream.sorted()`, *упорядочиваются*. Результаты накапливаются в порядке следования исходных элементов и полностью предсказуемы. Если выполнить одни и те же операции дважды, то будут получены совершенно одинаковые результаты.

Упорядочение не исключает эффективное распараллеливание. Например, при вызове `stream.map(fun)` поток данных может быть разбит на n сегментов, каждый из которых обрабатывается параллельно. А полученные результаты снова собираются по порядку.

Некоторые операции могут быть распараллелены более эффективно, если требование упорядочения опускается. Вызывая метод `Stream.unordered()`, можно указать, что упорядочение не имеет значения. Это, в частности, выгодно при выполнении операции методом `Stream.distinct()`. В упорядоченном потоке метод `distinct()` сохраняет первый из всех равных элементов. Этим ускоряется распараллеливание, поскольку в потоке исполнения, обрабатывающем отдельный сегмент, неизвестно, какие именно элементы следует отбросить, до тех пор, пока сегмент не будет обработан. Если же допускается сохранить

любой однозначный элемент, то все сегменты могут быть обработаны параллельно (с помощью общего множества для отслеживания дубликатов).

Если же опустить упорядочение, то можно ускорить выполнение метода `limit()`. А если требуется обработать любые *n* элементов из потока данных и при этом неважно, какие из них будут получены, то с этой целью можно сделать следующий вызов:

```
Stream<String> sample = words.parallelStream()
                             .unordered().limit(n);
```

Как обсуждалось в разделе 1.9, объединять отображения невыгодно из-за немалых затрат. Именно поэтому в методе `Collectors.groupingByConcurrent()` используется общее параллельное отображение. Чтобы извлечь выгоду из параллелизма, порядок следования значений в отображении должен быть иным, чем в потоке данных:

```
Map<Integer, List<String>> result =
    words.parallelStream().collect(
          Collectors.groupingByConcurrent(String::length));
          // Значения не накапливаются в потоковом порядке
```

Разумеется, это не имеет особого значения, если применяется нисходящий коллектор, не зависящий от упорядочения, как демонстрируется в следующем примере кода:

```
Map<Integer, Long> wordCounts =
    words.parallelStream()
         .collect(groupingByConcurrent(
                  String::length, counting()));
```

Не пытайтесь сделать все потоки данных параллельными, надеясь тем самым ускорить выполнение операций над ними. Вместо этого принимайте во внимание следующее.

- Распараллеливание требует немалых затрат, которые окупаются лишь при обработке очень крупных массивов данных.
- Распараллеливание потока данных дает преимущества лишь в том случае, если источник исходных данных удается эффективно разделить на несколько частей.
- Пул потоков исполнения, применяемый в параллельных потоках данных, может зависнуть при выполнении таких блокирующих операций, как ввод-вывод или доступ к сети.

Параллельные потоки данных лучше всего подходят для обработки крупных массивов данных в оперативной памяти с интенсивными вычислениями.

СОВЕТ. До версии Java 9 распараллеливать поток данных, возвращавшийся методом **`Files.lines()`**, не имело никакого смысла. Данные в таком потоке не подлежали разделению на части, поэтому приходилось читать сначала первую половину файла, а затем вторую. Теперь данный метод может оперировать отображаемым в памяти файлом, эффективно разделяя его содержимое на части. Так, если обрабатываются строки из крупного файла, то, распараллеливая поток данных, можно существенно повысить производительность.

НА ЗАМЕТКУ! По умолчанию в параллельных потоках данных применяется глобальный пул вилочного соединения, возвращаемый методом **ForkJoinPool.commonPool()**. И этого оказывается достаточно, если выполняемые операции не блокируются, а пул не требуется разделять как общий вместе с другими задачами. В противном случае придется употребить специальный прием, подставив другой пул. Для этого достаточно разместить выполняемые операции в теле метода **submit()** из специального пула следующим образом:

```
ForkJoinPool customPool = ...;
result = customPool.submit(() ->
         stream.parallel().map(...).collect(...)).get();
```

или же сделать то же самое, но асинхронно:

```
CompletableFuture.supplyAsync(() ->
           stream.parallel().map(...).collect(...),
            customPool).thenAccept(result -> ...);
```

НА ЗАМЕТКУ! Если требуется распараллелить операции над потоками данных на основании случайных чисел, этот процесс *не* следует начинать с потоков данных, возвращаемых методами **Random.ints()**, **Random.longs()** и **Random.doubles()**, поскольку такие потоки не разделяются на части. Вместо этого лучше воспользоваться методами **ints()**, **longs()** и **doubles()** из класса **SplittableRandom**.

В примере кода из листинга 1.8 демонстрируется, как следует оперировать параллельными потоками данных.

Листинг 1.8. Исходный код из файла **parallel/ParallelStreams.java**

```
1  package parallel;
2
3  /**
4   * @version 1.01 2018-05-01
5   * @author Cay Horstmann
6   */
7
8  import static java.util.stream.Collectors.*;
9
10 import java.io.*;
11 import java.nio.charset.*;
12 import java.nio.file.*;
13 import java.util.*;
14 import java.util.stream.*;
15
16 public class ParallelStreams
17 {
18   public static void main(String[] args)
19         throws IOException
20   {
21     var contents = new String(Files.readAllBytes(
22         Paths.get("../gutenberg/alice30.txt")),
23                   StandardCharsets.UTF_8);
24     List<String> wordList = List.of(contents.split("\\PL+"));
25
26     // ниже приведен очень скверный код:
```

```
27      var shortWords = new int[10];
28      wordList.parallelStream().forEach(s ->
29          {
30            if (s.length() < 10) shortWords[s.length()]++;
31          });
32      System.out.println(Arrays.toString(shortWords));
33
34      // попробовать снова, хотя результат вряд ли
35      // окажется иным, т.е. неверным
36      Arrays.fill(shortWords, 0);
37      wordList.parallelStream().forEach(s ->
38          {
39            if (s.length() < 10) shortWords[s.length()]++;
40          });
41      System.out.println(Arrays.toString(shortWords));
42
43      // выход: сгруппировать и подсчитать результаты
44      Map<Integer, Long> shortWordCounts = wordList
45         .parallelStream()
46         .filter(s -> s.length() < 10)
47         .collect(groupingBy(String::length, counting()));
48
49      System.out.println(shortWordCounts);
50
51      // нисходящий порядок не детерминирован:
52      Map<Integer, List<String>> result =
53        wordList.parallelStream().collect(
54                 Collectors.groupingByConcurrent(
55                           String::length));
56
57      System.out.println(result.get(14));
58
59      result = wordList.parallelStream().collect(
60         Collectors.groupingByConcurrent(String::length));
61
62      System.out.println(result.get(14));
63
64      Map<Integer, Long> wordCounts =
65          wordList.parallelStream().collect(
66                   groupingByConcurrent(
67                           String::length, counting()));
68
69      System.out.println(wordCounts);
70    }
71  }
```

java.util.stream.BaseStream<T, S extends BaseStream<T, S>> 8
• **S parallel()** Возвращает параллельный поток данных с такими же элементами, как и у исходного потока. • **S unordered()** Возвращает неупорядоченный поток данных с такими же элементами, как и у исходного потока.

java.util.Collection<E> 1.2

- **Stream<E> parallelStream()** 8
  Возвращает параллельный поток данных с элементами из исходной коллекции.

В этой главе было показано, как применять на практике библиотеку потоков данных, внедренную в версии Java 8. В следующей главе рассматривается не менее важная тема организации ввода-вывода.

ГЛАВА 2

Ввод и вывод

В этой главе...

- Потоки ввода-вывода
- Чтение и запись двоичных данных
- Потоки ввода-вывода и сериализация объектов
- Манипулирование файлами
- Файлы, отображаемые в памяти
- Регулярные выражения

В этой главе речь пойдет о прикладных интерфейсах (API), доступных в Java для организации ввода-вывода данных. Из нее вы, в частности, узнаете, как получать доступ к файлам и каталогам, как читать и записывать данные в двоичном и текстовом формате. В главе рассматривается также механизм сериализации, позволяющий сохранять объекты так же просто, как и текстовые или числовые данные. Далее речь пойдет о манипулировании файлами и каталогами. И в завершение обсуждаются регулярные выражения, хотя они и не имеют непосредственного отношения к потокам ввода-вывода и файлам. Тем не менее трудно найти более подходящее место для обсуждения этой темы, как его, очевидно, не удалось найти и разработчикам Java, присоединившим спецификацию прикладного интерфейса API для регулярных выражений к запросу на уточнение новых средств ввода-вывода.

2.1. Потоки ввода-вывода

В прикладном интерфейсе Java API объект, из которого можно читать последовательность байтов, называется *потоком ввода*, а объект, в который можно записывать последовательность байтов, — *потоком вывода*. В роли таких источников

и приемников последовательностей байтов чаще всего выступают файлы, но могут также служить сетевые соединения и даже блоки памяти. Абстрактные классы `InputStream` и `OutputStream` служат основанием для иерархии классов ввода-вывода.

НА ЗАМЕТКУ! Рассматриваемые здесь потоки ввода-вывода никак не связаны с потоками данных, представленными в предыдущей главе. Ради ясности они называются в тексте потоками ввода и вывода всякий раз, когда речь идет о вводе-выводе.

Байтовые потоки ввода-вывода неудобны для обработки информации, хранящейся в Юникоде (напомним, что в Юникоде на каждый символ приходится несколько байтов). Поэтому для обработки символов в Юникоде предусмотрена отдельная иерархия классов, наследующих от абстрактных классов `Reader` и `Writer`. Эти классы позволяют выполнять операции чтения и записи на основании двухбайтовых значений типа `char` (т.е. кодовых единиц в кодировке UTF-16), а не однобайтовых значений типа `byte`.

2.1.1. Чтение и запись байтов

В классе `InputStream` имеется следующий абстрактный метод:

```
abstract int read()
```

Этот метод читает один байт и возвращает считанный байт или значение `-1`, если обнаруживается конец источника ввода. Разработчик конкретного класса потока ввода может переопределить этот метод таким образом, чтобы он предоставлял какую-нибудь полезную функциональную возможность. Например, в классе `FileInputStream` этот метод выполняет чтение одного байта из файла. А поток ввода `System.in` представляет собой предопределенный объект подкласса `InputStream`, который позволяет читать данные из "стандартного ввода", т.е. консоли или переадресовываемого файла.

У класса `InputStream` имеются также неабстрактные методы для чтения массива байтов или пропуска определенного ряда байтов. В версии Java 9 в этом классе появился приведенный ниже очень удобный метод для чтения байтов из потока данных. Имеются также методы для чтения заданного количества байтов (см. примечания к прикладному интерфейсу API в конце этого раздела). В этих методах вызывается абстрактный метод `read()`, благодаря чему в подклассах достаточно переопределить только один метод.

```
byte[] bytes = in.readAllBytes();
```

Аналогично в классе `OutputStream` определяется следующий абстрактный метод, записывающий один байт в указанное место для вывода данных:

```
abstract void write(int b)
```

Если же имеется массив байтов, их можно записать все вместе, как показано ниже.

```
byte[] values = ...;
out.write(values);
```

Метод `transferTo()` направляет все байты из потока ввода в поток вывода следующим образом:

```
in.transferTo(out);
```

Как методы `read()`, так и методы `write()` *блокируют* доступ до тех пор, пока байты не будут фактически считаны или записаны. Это означает, что если к потоку ввода-вывода не удается получить доступ немедленно (что обычно случается из-за занятости сетевого соединения), происходит блокирование текущего потока исполнения. А это дает другим потокам исполнения возможность выполнять какую-нибудь полезную задачу, в то время как метод ожидает, когда поток ввода-вывода снова станет доступным.

Метод `available()` позволяет проверить количество байтов, доступное для считывания в текущий момент. Это означает, что фрагмент кода, аналогичный представленному ниже, вряд ли приведет к блокировке.

```
int bytesAvailable = in.available();
if (bytesAvailable > 0)
{
  var data = new byte[bytesAvailable];
  in.read(data);
}
```

По завершении чтения или записи данных в поток ввода-вывода следует закрыть его, вызвав метод `close()`. Такой вызов приводит к освобождению системных ресурсов, доступных в ограниченном количестве. Если же в прикладной программе открывается слишком много потоков ввода-вывода без последующего их закрытия, ресурсы системы могут исчерпаться. Кроме того, закрытие потока вывода приводит к *очистке* использовавшегося для него буфера: все байты, которые временно размещались в этом буфере с целью их последующей доставки в виде более крупного пакета, рассылаются по местам своего назначения. Так, если не закрыть файл, последний пакет байтов может так никогда и не быть доставлен. Очистить буфер от выводимых данных можно и вручную с помощью метода `flush()`.

Даже если в классе потока ввода-вывода предоставляются конкретные методы для работы с базовыми функциями чтения и записи, разработчики прикладных программ редко пользуются ими. Для них больший интерес представляют данные, содержащие числа, символьные строки и объекты, а не исходные байты. Поэтому в Java предоставляется немало классов потоков ввода-вывода, наследуемых от базовых классов `InputStream` и `OutputStream` и позволяющих обрабатывать данные в нужной форме, а не просто в виде исходных байтов.

`java.io.InputStream` 1.0

- **`abstract int read()`**

  Считывает байт данных и возвращает его. По достижении конца потока возвращает значение `-1`.

- **`int read(byte[] b)`**

  Считывает данные в байтовый массив и возвращает фактическое количество считанных байтов или значение **`-1`**, если достигнут конец потока ввода. Этот метод позволяет считать максимум ***`b.length`*** байтов.

java.io.InputStream 1.0 *(окончание)*

- **int read(byte[] *b*, int *off*, int *len*)**
- **int readNBytes(byte[] b, int *off*, int *len*)** 9

  Считывают количество байтов вплоть до ***len***, не блокируясь (метод **read()**) или же блокируясь (метод **readNBytes()**) до тех пор, пока не будут прочитаны все значения. Считанные значения размещаются в массиве **b**, начиная с позиции ***off***. Возвращают фактическое количество прочитанных байтов или значение **-1**, если достигнут конец потока ввода.
- **byte[] readAllBytes()** 9

  Возвращает массив всех байтов, которые могут быть прочитаны из данного потока ввода.
- **long transferTo(OutputStream *out*)** 9

  Направляет все байты из данного потока ввода в заданный поток вывода, возвращая количество направленных байтов. Ни тот, ни другой поток при этом не закрывается.
- **long skip(long *n*)**

  Пропускает ***n*** байтов в потоке ввода. Возвращает фактическое количество пропущенных байтов (которое может оказаться меньше ***n***, если достигнут конец потока ввода).
- **int available()**

  Возвращает количество байтов, доступных без блокирования. (Напомним, что блокирование означает потерю текущим потоком исполнения своей очереди на выполнение.)
- **void close()**

  Закрывает поток ввода.
- **void mark(int *readlimit*)**

  Устанавливает маркер на текущей позиции в потоке ввода. (Не все потоки поддерживают такую функциональную возможность.) Если из потока ввода считано байтов больше заданного предела ***readlimit***, в потоке ввода можно пренебречь устанавливаемым маркером.
- **void reset()**

  Возвращается к последнему маркеру. Последующие вызовы метода **read()** приводят к повторному считыванию байтов. В отсутствие текущего маркера поток ввода не устанавливается в исходное положение.
- **boolean markSupported()**

  Возвращает логическое значение **true**, если в потоке ввода поддерживается возможность устанавливать маркеры.

java.io.OutputStream 1.0

- **abstract void write(int *n*)**

  Записывает байт данных.
- **void write(byte[] *b*)**
- **void write(byte[] *b*, int *off*, int *len*)**

  Записывают все байты или определенный ряд байтов из массива ***b***.
- **void close()**

  Очищает и закрывает поток вывода.
- **void flush()**

  Очищает поток вывода, отправляя любые находящиеся в буфере данные по месту их назначения.

2.1.2. Полный комплект потоков ввода-вывода

В отличие от языка C, где для ввода-вывода достаточно и единственного типа `FILE*`, в Java имеется целый комплект из более чем 60 (!) различных типов потоков ввода-вывода (рис. 2.1 и 2.2). Разделим эти типы по областям их применения. Так, для классов, обрабатывающих байты и символы, существуют отдельные иерархии.

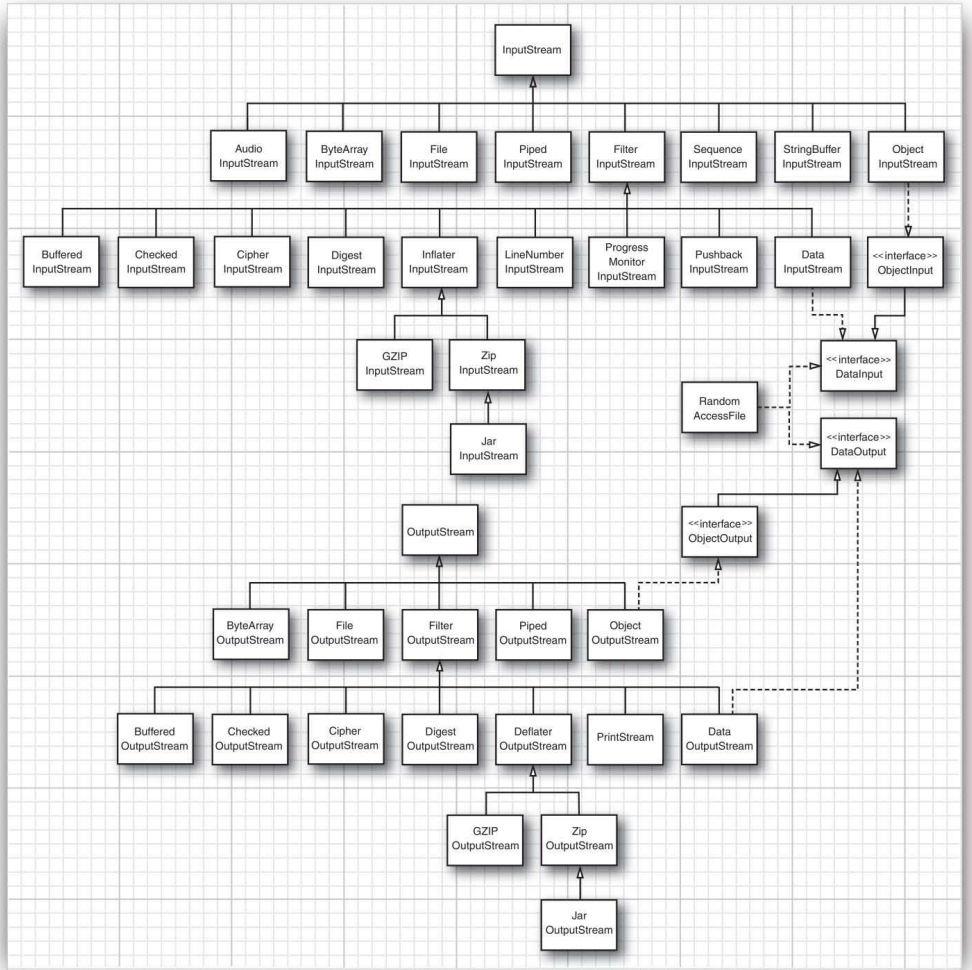


Рис. 2.1. Иерархия классов для потоков ввода-вывода

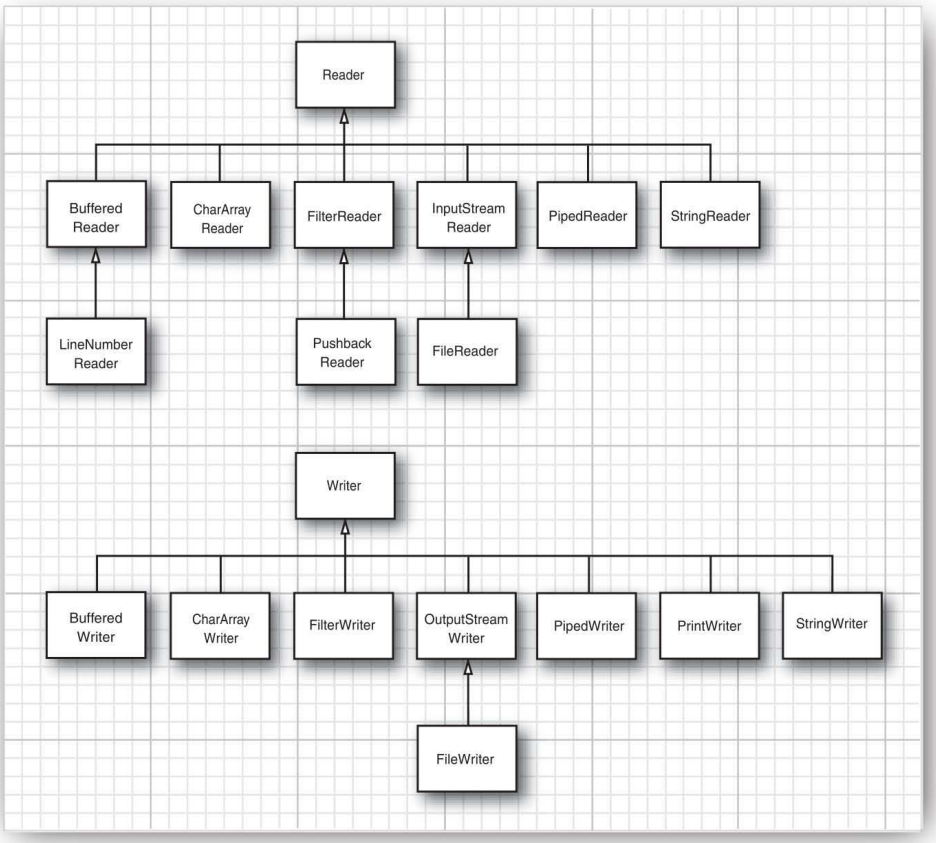


Рис. 2.2. Иерархия классов **`Reader`** и **`Writer`**

Как упоминалось выше, классы `InputStream` и `OutputStream` позволяют выполнять чтение и запись отдельных байтов и массивов байтов. Эти классы образуют основу иерархии, приведенной на рис. 2.1. Для чтения и записи символьных строк и чисел требуются подклассы, обладающие немалыми функциональными возможностями. Например, классы `DataInputStream` и `DataOutputStream` позволяют выполнять чтение и запись всех простых типов данных Java в двоичном формате. И, наконец, имеются классы для выполнения отдельных полезных операций ввода-вывода, например, классы `ZipInputStream` и `ZipOutputStream`, позволяющие читать и записывать данные в файлы с уплотнением в таком известном формате, как ZIP.

С другой стороны, для ввода-вывода текста в Юникоде необходимо обращаться к подклассам таких абстрактных классов, как `Reader` и `Writer` (см. рис. 2.2). Базовые методы из классов `Reader` и `Writer` похожи на базовые методы из классов `InputStream` и `OutputStream`, как показано ниже.

```
abstract int read()
abstract void write(int c)
```

Метод `read()` возвращает кодовую единицу в Юникоде (в виде целого числа от `0` до `65535`) или значение `-1`, если достигнут конец файла. А метод `write()` вызывается с заданной кодовой единицей в Юникоде. Подробнее о кодовых единицах в частности и Юникоде вообще см. в главе 3 первого тома настоящего издания.

Имеются также четыре дополнительных интерфейса: `Closeable`, `Flushable`, `Readable` и `Appendable` (рис. 2.3). Первые два очень просты: в них определяется единственный метод `void close() throws IOException` и `void flush()` соответственно. Классы `InputStream`, `OutputStream`, `Reader` и `Writer` реализуют интерфейс `Closeable`, а классы `OutputStream` и `Writer` — интерфейс `Flushable`.

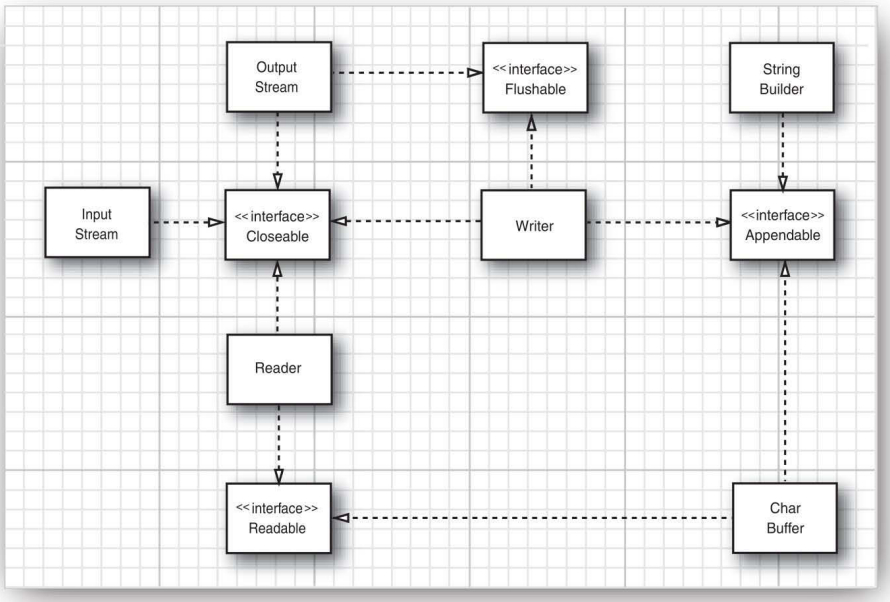


Рис. 2.3. Интерфейсы **`Closeable`**, **`Flushable`**, **`Readable`** и **`Appendable`**

НА ЗАМЕТКУ! Интерфейс **`java.io.Closeable`** расширяет интерфейс **`java.lang.AutoCloseable`**. Следовательно, в любой реализации интерфейса **`Closeable`** можно употреблять оператор **`try`** с ресурсами. А зачем вообще нужны два интерфейса? В методе **`close()`** из интерфейса **`Closeable`** генерируется только исключение типа **`IOException`**, тогда как в методе **`AutoCloseable.close()`** исключение может быть вообще не сгенерировано.

В интерфейсе `Readable` имеется единственный метод `int read(CharBuffer cb)`, а в классе `CharBuffer` — методы для чтения и записи с последовательным и произвольным доступом. Этот класс представляет буфер в оперативной памяти или отображаемый в памяти файл. (Подробнее об этом речь пойдет далее, в разделе 2.5.2.)

В интерфейсе `Appendable` имеются два приведенных ниже метода, позволяющие присоединять как отдельные символы, так и целые последовательности символов.

```
Appendable append(char c)
Appendable append(CharSequence s)
```

Интерфейс `CharSequence` описывает основные свойства последовательности значений типа `char`. Его реализуют такие классы, как `String`, `CharBuffer`, `StringBuilder` и `StringBuffer`. Из всех классов, представляющих потоки ввода-вывода, только класс `Writer` реализует интерфейс `Appendable`.

java.io.Closeable 5.0

- **void close()**
  Закрывает данный поток ввода-вывода типа **Closeable**. Этот метод может генерировать исключение типа **IOException**.

java.io.Flushable 5.0

- **void flush()**
  Очищает данный поток ввода-вывода типа **Flushable**.

java.lang.Readable 5.0

- **int read(CharBuffer *cb*)**
  Пытается считать столько значений типа **char** в буфер ***cb***, сколько в нем может их уместиться. Возвращает количество считанных значений типа **char** или значение **-1**, если из данного потока ввода типа **Readable** больше не доступно никаких значений.

java.lang.Appendable 5.0

- **Appendable append(char *c*)**
- **Appendable append(CharSequence *cs*)**
  Присоединяют указанную кодовую единицу или все кодовые единицы из заданной последовательности к данному потоку ввода-вывода типа **Appendable**. Возвращают ссылку **this**.

java.lang.CharSequence 1.4

- **char charAt(int *index*)**
  Возвращает кодовую единицу по заданному индексу.
- **int length()**
  Возвращает сведения об общем количестве кодовых единиц в данной последовательности.
- **CharSequence subSequence(int *startIndex*, int *endIndex*)**
  Возвращает последовательность типа **CharSequence**, состоящую только из тех кодовых единиц, которые хранятся в пределах от ***startIndex*** до ***endIndex*** **- 1**.
- **String toString()**
  Возвращает символьную строку, состоящую только из тех кодовых единиц, которые входят в данную последовательность.

2.1.3. Сочетание фильтров потоков ввода-вывода

Классы FileInputStream и FileOutputStream позволяют создавать потоки ввода-вывода и присоединять их к конкретному файлу на диске. Имя требуемого файла и полный путь к нему указываются в конструкторе соответствующего класса. Например, в приведенной ниже строке кода поиск файла employee.dat будет производиться в каталоге пользователя.

```
FileInputStream fin = new FileInputStream("employee.dat");
```

СОВЕТ. Во всех классах из пакета **java.io** относительные пути к файлам воспринимаются как команда начинать поиск с рабочего каталога пользователя, поэтому может возникнуть потребность выяснить содержимое этого каталога. Для этого достаточно вызвать метод **System.getProperty("user.dir")**.

ВНИМАНИЕ! Знак обратной косой черты является экранирующим в символьных строках Java, поэтому указывайте в путях к файлам по два таких знака подряд, как, например, **C:\\Windows\\win.ini**. В Windows допускается указывать в путях к файлам одиночные знаки прямой (или просто) косой черты, как, например, **C:/Windows/win.ini**, поскольку в большинстве вызовов из файловой системы Windows знаки косой черты будут интерпретироваться как разделители файлов. Но делать это все же не рекомендуется, поскольку в режиме работы файловой системы Windows возможны изменения. Вместо этого ради переносимости программ в качестве разделителя файлов лучше употреблять именно тот знак, который принят на данной платформе. Такой знак доступен в виде строковой константы **java.io.File.separator**.

Как и в абстрактных классах InputStream и OutputStream, в классах FileInputStream и FileOutputStream поддерживаются чтение и запись только на уровне байтов. Так, из объекта fin в приведенной ниже строке кода можно только считать отдельные байты и массивы байтов.

```
byte b = (byte) fin.read();
```

Как поясняется в следующем разделе, имея в своем распоряжении только класс DataInputStream, можно было бы читать данные числовых типов следующим образом:

```
DataInputStream din = ...;
double s = din.readDouble();
```

Но, как и в классе FileInputStream отсутствуют методы для чтения данных числовых типов, так и в классе DataInputStream отсутствуют методы для извлечения данных из файла.

В языке Java применяется искусный механизм для разделения двух видов обязанностей. Одни потоки ввода-вывода (типа FileInputStream и поток ввода, возвращаемый методом openStream() из класса URL) могут извлекать байты из файлов и других экзотических мест, а другие потоки ввода-вывода (типа DataInputStream) — составлять эти байты в более полезные типы данных. Программирующему на Java остается только употреблять их в нужном сочетании. Например, чтобы получить возможность читать числа из файла, достаточно создать сначала объект потока ввода типа FileInputStream, а затем передать его конструктору класса DataInputStream, как показано ниже.

```
var fin = new FileInputStream("employee.dat");
var din = new DataInputStream(fin);
double x = din.readDouble();
```

Если снова обратиться к рис. 2.1, то в иерархии классов для потоков ввода-вывода можно обнаружить классы `FilterInputStream` и `FilterOutputStream`. Подклассы этих классов служат для расширения функциональных возможностей потоков ввода-вывода, обрабатывающих байты.

Благодаря вложению фильтров функциональные возможности потоков ввода-вывода удается расширить в еще большей степени. Например, по умолчанию потоки ввода не буферизуются. Это означает, что каждый вызов метода `read()` приводит к запрашиванию у операционной системы выдачи очередного байта. Но намного эффективнее запрашивать сразу целые блоки данных и размещать их в буфере. Потребность использовать буферизацию *и* методы ввода данных в файл диктует применение следующей довольно громоздкой последовательности конструкторов:

```
var din = new DataInputStream(
    new BufferedInputStream(
        new FileInputStream("employee.dat")));
```

Обратите внимание на то, что конструктор класса `DataInputStream` указывается *последним* в цепочке конструкторов. Ведь в данном случае предполагается использовать методы из класса `DataInputStream`, а в них — буферизуемый метод `read()`.

Иногда возникает потребность отслеживать промежуточные потоки ввода-вывода, когда они соединяются в цепочку. Например, при чтении входных данных нередко требуется считывать следующий байт с упреждением, чтобы выяснить, содержится ли в нем предполагаемое значение. Для этой цели в Java предоставляется класс `PushbackInputStream`, как показано ниже.

```
var pbin = new PushbackInputStream(
           new BufferedInputStream(
               new FileInputStream("employee.dat")));
```

Теперь можно прочитать сначала следующий байт с упреждением:

```
int b = pbin.read();
```

а затем возвратить его обратно, если он не содержит именно то, что нужно:

```
if (b != '<') pbin.unread(b);
```

Но методы чтения `read()` и непрочтения `unread()` являются *единственными*, которые можно применять в потоке ввода типа `PushbackInputStream`. Так, если нужно считывать числовые данные с упреждением, то для этого потребуется ссылка не только на поток ввода типа `PushBackInputStream`, но и на поток ввода типа `DataInputStream`, как выделено ниже полужирным.

```
var din = new DataInputStream(
   pbin = new PushbackInputStream(
          new BufferedInputStream(
              new FileInputStream("employee.dat"))));
```

Безусловно, в библиотеках потоков ввода-вывода на других языках программирования такие полезные функции, как буферизация и чтение с упреждением,

обеспечиваются автоматически, и поэтому в Java необходимость сочетать для их реализации потоковые фильтры доставляет лишние хлопоты. Но в то же время возможность сочетать и подбирать классы фильтров для создания действительно полезных последовательностей потоков ввода-вывода дает немалую свободу действий. Например, используя приведенную ниже последовательность потоков ввода, можно организовать чтение чисел из архивного файла, уплотненного в формате ZIP (рис. 2.4). (Более подробно о том, как манипулировать в Java файлами, уплотненными в формате ZIP, речь пойдет в разделе 2.2.3.)

```
var zin = new ZipInputStream(
          new FileInputStream("employee.zip"));
var din = new DataInputStream(zin);
```

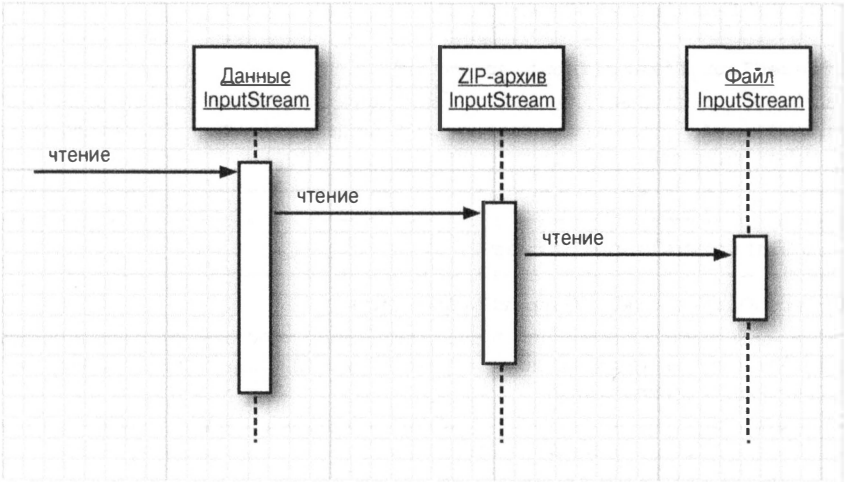


Рис. 2.4. Последовательность фильтруемых потоков

java.io.FileInputStream 1.0

- **FileInputStream(String *name*)**
- **FileInputStream(File *file*)**

  Создают новый поток ввода из файла, путь к которому указывается в символьной строке ***name*** или в объекте ***file***. (Более подробно класс **File** описывается в конце этой главы.) Если указываемый путь не является абсолютным, он определяется относительно рабочего каталога, который был установлен при запуске виртуальной машины.

java.io.FileOutputStream 1.0

- **FileOutputStream(String *name*)**
- **FileOutputStream(String *name*, boolean *append*)**

java.io.FileOutputStream 1.0 *(окончание)*

- **FileOutputStream(File *file*)**
- **FileOutputStream(File *file*, boolean *append*)**

  Создают новый поток вывода в файл, который указывается в символьной строке ***name*** или в объекте ***file***. (Более подробно класс **File** описывается в конце этой главы.) Если параметр ***append*** принимает логическое значение **true**, существующий файл с таким же именем не удаляется, а данные добавляются в конце файла. В противном случае удаляется любой уже существующий файл с таким же именем.

java.io.BufferedInputStream 1.0

- **BufferedInputStream(InputStream *in*)**

  Создает буферизированный поток ввода. Такой поток способен накапливать вводимые байты без постоянного обращения к устройству ввода. Когда буфер пуст, в него считывается новый блок данных из потока.

java.io.BufferedOutputStream 1.0

- **BufferedOutputStream(OutputStream *out*)**

  Создает буферизированный поток вывода. Такой поток способен накапливать выводимые байты без постоянного обращения к устройству вывода. Когда буфер заполнен или поток очищен, данные записываются.

java.io.PushbackInputStream 1.0

- **PushbackInputStream(InputStream *in*)**
- **PushbackInputStream(InputStream *in*, int *size*)**

  Создают поток ввода или вывода с однобайтовым буфером для чтения с упреждением или буфером указанного размера для возврата данных обратно в поток.
- **void unread(int *b*)**

  Возвращает байт обратно в поток, чтобы он мог быть извлечен снова при последующем вызове для чтения.

2.1.4. Ввод-вывод текста

При сохранении данных приходится выбирать между двоичным и текстовым форматом. Так, если целое число `1234` сохраняется в двоичном формате, оно записывается в виде следующей последовательности байтов: `00 00 04 D2` (в шестнадцатеричном представлении), а в текстовом формате — в виде символьной строки `"1234"`. Хотя ввод-вывод данных в двоичном формате осуществляется быстрее и эффективнее, тем не менее, они неудобочитаемы. Поэтому мы обсудим сначала ввод-вывод текстовых данных, а затем двоичных (в разделе 2.2).

При сохранении текстовых строк приходится учитывать конкретную *кодировку символов*. Так, если используется кодировка UTF-16, символьная строка `"José"`

кодируется последовательностью байтов 00 4A 00 6F 00 73 00 E9 (в шестнадцатеричном представлении). Но во многих прикладных программах предполагается другая кодировка содержимого текстовых файлов. Так, в кодировке UTF-8, чаще всего употребляемой в Интернете, упомянутая выше символьная строка будет записана в виде последовательности байтов 4A 6F 73 C3 A9, где первые три буквы представлены без нулевых байтов, а последняя буква é — двумя байтами.

Класс `OutputStreamWriter` превращает поток вывода кодовых единиц Юникода в поток записи байтов, применяя выбранную кодировку символов, а класс `InputStreamReader`, наоборот, — превращает поток ввода байтов, представляющих символы в какой-нибудь кодировке, в поток чтения, выдающий символы в виде кодовых единиц Юникода.

В качестве примера ниже показано, как создать поток чтения вводимых данных, способный считывать с консоли набираемые на клавиатуре символы и преобразовывать их в Юникод.

```
InputStreamReader in = new InputStreamReader(System.in);
```

В этом потоке чтения вводимых данных предполагается, что в системе используется стандартная кодировка символов. В настольных операционных системах может применяться устаревшая кодировка вроде Windows 1252 или MacRoman. Но ничто не мешает выбрать любую другую кодировку, указав ее в конструкторе класса `InputStreamReader`, например, так, как показано ниже. Подробнее кодировки символов будут обсуждаться далее, в разделе 2.1.8.

```
var in = new InputStreamReader(new FileInputStream(
            "data.txt"), StandardCharsets.UTF_8);
```

2.1.5. Вывод текста

Для вывода текста лучше всего подходит класс `PrintWriter`. В этом классе имеются методы для вывода символьных строк и чисел в текстовом формате. Чтобы вывести текст в файл, достаточно создать объект типа `PrintWriter`, указав имя этого файла, а также конкретную кодировку, как показано ниже.

```
var out = new PrintWriter("employee.txt",
                          StandardCharsets.UTF_8);
```

Для вывода текста в поток записи типа `PrintWriter` применяются те же методы `print()`, `println()` и `printf()`, что и для вывода в стандартный поток `System.out`. Эти методы можно использовать для вывода числовых данных (типа `int`, `short`, `long`, `float`, `double`), символов, логических значений типа `boolean`, символьных строк и объектов.

Рассмотрим в качестве примера следующий фрагмент кода:

```
String name = "Harry Hacker";
double salary = 75000;
out.print(name);
out.print(' ');
out.println(salary);
```

В этом фрагменте кода текстовая строка `"Harry Hacker 75000.0"` выводится в поток `out`, а затем преобразуется в байты и в конечном итоге записывается в файл `employee.txt`.

Метод `println()` добавляет к ней символ конца строки, подходящий для целевой платформы (`"\r\n"` — для Windows, `"\n"` — для Unix). А получается этот символ конца строки в результате вызова `System.getProperty("line.separator")`.

Если поток записи выводимых данных устанавливается в режим *автоматической очистки*, то при каждом вызове метода `println()` все символы, хранящиеся в буфере, отправляются по месту их назначения. (Потоки записи выводимых данных всегда снабжаются буфером.) По умолчанию режим автоматической очистки *не* включается. Его можно включать и выключать с помощью конструктора `PrintWriter(Writer out, boolean autoFlush)` следующим образом:

```
var out = new PrintWriter(
new OutputStreamWriter(
    new FileOutputStream("employee.txt"),
    StandardCharsets.UTF_8), true); // автоочистка
```

Методы типа `print` не генерируют исключений. Чтобы проверить наличие ошибок в потоке вывода, следует вызывать метод `checkError()`.

НА ЗАМЕТКУ! У программирующих на Java со стажем может возникнуть следующий вопрос: что же случилось с классом **`PrintStream`** и стандартным потоком вывода **`System.out`**? В версии Java 1.0 класс **`PrintStream`** просто усекал все символы Юникода до символов в коде ASCII, отбрасывая старший байт (в то время в Юникоде еще применялась 16-разрядная кодировка). Очевидно, что такой подход не обеспечивал точность и переносимость результатов, из-за чего он был усовершенствован внедрением в версии Java 1.1 потоков чтения и записи данных. Для обеспечения совместимости с существующим кодом **`System.in`**, **`System.out`** и **`System.err`** по-прежнему являются потоками ввода-вывода, но не для чтения и записи данных.

Класс **`PrintStream`** теперь способен преобразовывать внутренним образом символы Юникода в стандартную кодировку хоста точно так же, как и класс **`PrintWriter`**. Объекты типа **`PrintStream`** действуют таким же образом, как и объекты типа **`PrintWriter`**, когда вызываются методы **`print()`** и **`println()`**, но, в отличие от объектов типа **`PrintWriter`**, они позволяют также выводить исходные байты с помощью методов **`write(int)`** и **`write(byte[])`**.

`java.io.PrintWriter` 1.1

- **`PrintWriter(Writer out)`**
- **`PrintWriter(Writer writer)`**

  Создают новый объект типа **`PrintWriter`**, выводящий данные в указанный поток записи.
- **`PrintWriter(String filename, String encoding)`**
- **`PrintWriter(File file, String encoding)`**

  Создают новый объект типа **`PrintWriter`**, выводящий данные в заданный файл, используя указанную кодировку.
- **`void print(Object obj)`**

  Выводит объект в виде символьной строки, получаемой из метода **`toString()`**.
- **`void print(String s)`**

  Выводит символьную строку в виде кодовых единиц Юникода.

java.io.PrintWriter 1.1 *(окончание)*

- **void println(String *s*)**
  Выводит символьную строку вместе с символом окончания строки. Очищает поток вывода, если он действует в режиме автоматической очистки.
- **void print(char[] *s*)**
  Выводит все символы из указанного массива в виде кодовых единиц Юникода.
- **void print(char *c*)**
  Выводит символ в виде кодовой единицы Юникода.
- **void print(int *i*)**
- **void print(long *l*)**
- **void print(float *f*)**
- **void print(double *d*)**
- **void print(boolean *b*)**
  Выводят заданное значение в текстовом формате.
- **void printf(String *format*, Object... *args*)**
  Выводит заданные значения так, как указано в форматирующей строке. О том, как задается форматирующая строка, см. в главе 3 первого тома настоящего издания.
- **boolean checkError()**
  Возвращает логическое значение **true**, если возникла ошибка при форматировании или выводе данных. При возникновении ошибки поток вывода считается испорченным, а в результате всех вызовов метода **checkError()** возвращается логическое значение **true**.

2.1.6. Ввод текста

Для обработки произвольно вводимого текста проще всего воспользоваться классом `Scanner`, как неоднократно демонстрировалось в примерах кода из первого тома настоящего издания. Поток сканирования типа `Scanner` можно построить из любого потока ввода.

С другой стороны, прочитать короткий текст из файла в символьную строку можно следующим образом:

```
String content =
       new String(Files.readAllBytes(path), charset);
```

Но если требуется прочитать содержимое файла в виде последовательности строк, необходимо сделать следующий вызов:

```
List<String> lines = Files.readAllLines(path, charset);
```

Если же файл крупный, строки можно обрабатывать по требованию в виде потока типа `Stream<String>`, как показано ниже.

```
try (Stream<String> lines = Files.lines(path, charset))
{
   ...
}
```

Поток сканирования можно также использовать для чтения *лексем* — символьных строк с разделителями. По умолчанию в качестве разделителей выбираются

пробелы, но их можно заменить любым регулярным выражением, как демонстрируется в следующем примере кода:

```
Scanner in = ...;
in.useDelimiter("\\PL+");
```

В данном случае в качестве разделителей лексем могут служить любые символы, кроме тех, которые входят в набор символов в Юникоде. Таким образом, поток сканирования будет принимать лексемы, состоящие только из символов в Юникоде.

В результате вызова метода `next()` возвращается следующая лексема:

```
while (in.hasNext())
{
  String word = in.next();
  ...
}
```

С другой стороны, поток всех лексем можно получить следующим образом:

```
Stream<String> words = in.tokens();
```

В ранних версиях Java единственным возможным вариантом для обработки вводимых текстовых данных был класс `BufferedReader`. В этом классе имеется метод `readLine()`, возвращающий текстовую строку или пустое значение `null`, если больше нечего вводить. Следовательно, типичный цикл ввода текстовых данных выглядит следующим образом:

```
InputStream inputStream = ...;
try (var in = new BufferedReader(
     new InputStreamReader(inputStream, charset)))
{
  String line;
  while ((line = in.readLine()) != null)
  {
    сделать что-нибудь с содержимым переменной line
  }
}
```

Ныне в классе `BufferedReader` появился также метод `lines()`, возвращающий поток типа `Stream<String>`. Но, в отличие от класса `Scanner`, в классе `BufferedReader` отсутствуют методы для чтения числовых данных.

2.1.7. Сохранение объектов в текстовом формате

В этом разделе рассматривается пример программы, сохраняющей массив записей типа `Employee` в текстовом файле. Каждая запись сохраняется в отдельной строке, поля отделяются один от другого разделителем. В качестве этого разделителя в данном примере используется вертикальная черта (`|`). (Другим распространенным разделителем является двоеточие (`:`). Любопытно, что каждый разработчик обычно пользуется своим разделителем.) Мы пока что не будем касаться того, что может произойти, если символ `|` встретится непосредственно в одной из сохраняемых символьных строк. Ниже приведен образец сохраняемых записей.

```
Harry Hacker|35500|1989-10-1
Carl Cracker|75000|1987-12-15
Tony Tester|38000|1990-03-15
```

Процесс записи происходит очень просто. Для этой цели применяется класс `PrintWriter`, поскольку запись выполняется в текстовый файл. Все поля записываются в файл, завершаясь символом `|`, а если это последнее поле, то комбинацией символов `\n`. Весь процесс записи совершается в теле приведенного ниже метода `writeData()`, который вводится в класс `Employee`.

```
public static void writeEmployee(PrintWriter out, Employee e)
{
  out.println(e.getName() + "|" + e.getSalary()
           + "|" + e.getHireDay());
}
```

Что касается чтения записей, то оно выполняется построчно с разделением полей. Для чтения каждой строки служит поток сканирования (типа `Scanner`), а затем полученная строка разбивается на лексемы с помощью метода `String.split()`, как показано ниже.

```
public static Employee readEmployee(Scanner in)
{
  String line = in.nextLine();
  String[] tokens = line.split("\\|");
  String name = tokens[0];
  double salary = Double.parseDouble(tokens[1]);
  LocalDate hireDate = LocalDate.parse(tokens[2]);
  int year = hireDate.getYear();
  int month = hireDate.getMonthValue();
  int day = hireDate.getDayOfMonth();
  return new Employee(name, salary, year, month, day);
}
```

В качестве параметра метода `split()` служит регулярное выражение, описывающее разделитель. Более подробно регулярные выражения рассматриваются в конце этой главы. Оказывается, что знак вертикальной черты (`|`) имеет в регулярных выражениях специальное значение, поэтому он должен обязательно экранироваться знаком `\`, а тот, в свою очередь, еще одним знаком `\`, в результате чего получается следующее регулярное выражение: `"\\|"`.

Весь исходный код данного примера программы представлен в листинге 2.1. А в приведенном ниже статическом методе сначала записывается длина массива, а затем — каждая запись.

```
void writeData(Employee[] e, PrintWriter out)
```

В следующем статическом методе сначала считывается длина массива, а затем каждая запись:

```
Employee[] readData(BufferedReader in)
```

Оказывается, что сделать это не так-то просто, как следует из приведенного ниже фрагмента кода.

```
int n = in.nextInt();
in.nextLine(); // перевести на новую строку
var employees = new Employee[n];
for (int i = 0; i < n; i++)
{
  employees[i] = new Employee();
  employees[i].readData(in);
}
```

Вызов метода `nextInt()` приводит к считыванию длины массива, но не завершающего символа новой строки. Этот символ должен обязательно употребляться, чтобы в методе `readData()` можно было перейти к следующей строке вводимых данных при вызове метода `nextLine()`.

Листинг 2.1. Исходный код из файла **textFile/TextFileTest.java**

```
1  package textFile;
2
3  import java.io.*;
4  import java.nio.charset.*;
5  import java.time.*;
6  import java.util.*;
7
8  /**
9   * @version 1.15 2018-03-17
10  * @author Cay Horstmann
11  */
12 public class TextFileTest
13 {
14   public static void main(String[] args)
15         throws IOException
16   {
17     var staff = new Employee[3];
18
19     staff[0] = new Employee("Carl Cracker",
20                             75000, 1987, 12, 15);
21     staff[1] = new Employee("Harry Hacker",
22                             50000, 1989, 10, 1);
23     staff[2] = new Employee("Tony Tester", 40000,
24                             1990, 3, 15);
25
26     // сохранить все записи о работниках
27     // в файле employee.dat
28     try (var out = new PrintWriter("employee.dat",
29                       StandardCharsets.UTF_8))
30     {
31        writeData(staff, out);
32     }
33
34     // извлечь все записи в новый массив
35     try (var in = new Scanner(new FileInputStream(
36                       "employee.dat"), "UTF-8"))
37     {
38       Employee[] newStaff = readData(in);
39
40       // вывести вновь прочитанные записи о работниках
41       for (Employee e : newStaff)
42         System.out.println(e);
43     }
44   }
45
46   /**
47    * Записывает данные обо всех работниках из
48    * массива в поток записи выводимых данных
```

```
49     * @param employees Массив записей о работниках
50     * @param out Поток записи выводимых данных
51     */
52    private static void writeData(Employee[] employees,
53                   PrintWriter out) throws IOException
54    {
55      // записать количество работников
56      out.println(employees.length);
57
58      for (Employee e : employees)
59        writeEmployee(out, e);
60    }
61
62    /**
63     * Читает записи о работниках из потока
64     * сканирования в массив
65     * @param in Поток сканирования вводимых данных
66     * @return Массив записей о работниках
67     */
68    private static Employee[] readData(Scanner in)
69    {
70      // извлечь размер массива
71      int n = in.nextInt();
72      in.nextLine(); // перевести на новую строку
73
74      var employees = new Employee[n];
75      for (int i = 0; i < n; i++)
76      {
77        employees[i] = readEmployee(in);
78      }
79       return employees;
80    }
81
82    /**
83     * Направляет данные о работниках в поток
84     * записи выводимых данных
85     * @param out Поток записи выводимых данных
86     */
87    public static void writeEmployee(
88                        PrintWriter out, Employee e)
89    {
90      out.println(e.getName() + "|" + e.getSalary()
91                + "|" + e.getHireDay());
92    }
93
94    /**
95     * Считывает данные о работниках из буферизованного
96     * потока чтения вводимых данных
97     * @param in Поток сканирования вводимых данных
98     */
99    public static Employee readEmployee(Scanner in)
100   {
101     String line = in.nextLine();
102     String[] tokens = line.split("\\|");
103     String name = tokens[0];
104     double salary = Double.parseDouble(tokens[1]);
105     LocalDate hireDate = LocalDate.parse(tokens[2]);
```

```
106     int year = hireDate.getYear();
107     int month = hireDate.getMonthValue();
108     int day = hireDate.getDayOfMonth();
109     return new Employee(name, salary, year, month, day);
110   }
111 }
```

2.1.8. Кодировки символов

Потоки ввода-вывода представляют собой последовательности байтов, но зачастую приходится обрабатывать текст, т.е. последовательности символов. В таком случае имеет значение, каким образом символы кодируются в байты.

Для кодирования символов в Java применяется стандарт, называемый *Юникодом* (Unicode). Каждый символ, или так называемая *кодовая точка*, представлен в Юникоде 21-разрядным целым числом. Имеются разные *кодировки символов* — способы упаковки 21-разрядных целых чисел в байты.

Чаще всего применяется кодировка UTF-8, в которой каждая кодовая точка в Юникоде кодируется последовательностью от одного до четырех байтов (табл. 2.1). Преимущество кодировки UTF-8 состоит в том, что символы из традиционного набора в коде ASCII, куда входят все буквы латинского и английского алфавитов, занимают только один байт.

Таблица 2.1. Кодировка UTF-8

Диапазон символов	Кодировка
0...7F	$0a_6a_5a_4a_3a_2a_1a_0$
80...7FF	$110a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$
800...FFFF	$1110a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$
10000...10FFFF	$11110a_{20}a_{19}a_{18}$ $10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$

Еще одной распространенной является кодировка UTF-16, в которой каждая кодовая точка в Юникоде кодируется одним или двумя 16-разрядными значениями (табл. 2.2). Такая кодировка применяется в символьных строках Java. На самом деле имеются две формы кодировки UTF-16: с обратным и прямым порядком следования байтов. Рассмотрим в качестве примера 16-разрядное значение `0x2122`. В формате с обратным порядком байтов первым следует старший байт `0x21`, а за ним — младший байт `0x22`. В формате с прямым порядком байтов все происходит наоборот: сначала следует младший байт `0x22`, а затем старший байт `0x21`. Для обозначения используемого порядка следования байтов файл может начинаться с соответствующей метки в виде 16-разрядного значения `0xFEFF`. С помощью этой метки пользователь может определить порядок следования байтов и отбросить его.

Таблица 2.2. Кодировка UTF-16

Диапазон символов	Кодировка
0...FFFF	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$
10000...10FFFF	$110110b_{19}b_{18}b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}$ $110111a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$, где $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$

ВНИМАНИЕ! В некоторых прикладных программах, в том числе в текстовом редакторе Microsoft Notepad, метка порядка следования байтов вводится в начале файлов, содержимое которых представлено в кодировке UTF-8. Очевидно, что это излишне, поскольку в кодировке UTF-8 вопросы, связанные с порядком следования байтов, не возникают. Тем не менее стандарт на Юникод допускает наличие такой метки и даже считает это целесообразным, поскольку она разрешает всякие сомнения по поводу кодировки. При чтении содержимого файла в кодировке UTF-8 такая метка должна удаляться. К сожалению, в Java этого не делается, а в отчетах об устраненных программных ошибках напротив данной ошибки стоит метка "не устранится". Поэтому любой начальный код **\uFEFF**, обнаруживаемый при вводе данных, придется удалять вручную.

Помимо упомянутых выше кодировок UTF, имеются частичные кодировки, охватывающие диапазон символов, пригодный для конкретного круга пользователей. Например, кодировка по стандарту ISO 8859-1 определяет однобайтовый код, включающий в себя символы с ударениями, применяемые в западноевропейских языках, а кодировка Shift-JIS — код переменной длины для японских символов. Немалое число подобных кодировок по-прежнему широко распространено.

Надежного способа автоматически выявить кодировку символов в потоке ввода байтов не существует. В некоторых методах из прикладного интерфейса API допускается применение "набора символов по умолчанию" — кодировки символов, которая считается наиболее предпочтительной в операционной системе компьютера. Но применяется ли та же самая кодировка и в источнике байтов? Ведь эти байты вполне могут поступать из разных частей света. Следовательно, кодировка символов должна всегда указываться явно. Так, при чтении веб-страницы следует проверять заголовок `Content-Type`.

НА ЗАМЕТКУ! Кодировка, принятая на конкретной платформе, возвращается статическим методом **`Charset.defaultCharset()`**. А статический метод **`Charset.availableCharsets()`** возвращает все имеющиеся экземпляры класса **`Charset`**, преобразованные из канонических имен в объекты типа **`Charset`**.

ВНИМАНИЕ! В реализации библиотеки Java от компании Oracle имеется системное свойство **`file.encoding`** для переопределения кодировки символов, принятой на конкретной платформе по умолчанию. Но это свойство не поддерживается официально и не последовательно согласуется со всеми частями реализации библиотеки Java от компании Oracle. Поэтому устанавливать его не следует.

В классе `StandardCharsets` имеются следующие статические переменные типа `Charset` для тех кодировок, которые должны поддерживаться на каждой виртуальной машине Java:

```
StandardCharsets.UTF_8
StandardCharsets.UTF_16
StandardCharsets.UTF_16BE
StandardCharsets.UTF_16LE
StandardCharsets.ISO_8859_1
StandardCharsets.US_ASCII
```

Чтобы получить объект типа `Charset` для другой кодировки, достаточно вызвать статический метод `forName()` следующим образом:

```
Charset shiftJIS = Charset.forName("Shift-JIS");
```

Для ввода-вывода текста следует пользоваться объектом типа `Charset`. В следующем примере кода показано, как превратить массив байтов в символьную строку:

```
var str = new String(bytes, StandardCharsets.UTF_8);
```

СОВЕТ. Начиная с версии Java 10, все методы из пакета **`java.io`** позволяют указывать конкретную кодировку символов с помощью объекта типа **`Charset`** или символьной строки. Следует также выбрать константы из класса **`StandardCharsets`**, чтобы перехватывать любые орфографические ошибки во время компиляции.

ВНИМАНИЕ! В одних методах и конструкторах, например, в конструкторе **`String(byte[])`**, используется кодировка, принятая на платформе по умолчанию, если не указано иное. А в других методах и конструкторах, например, в конструкторе **`Files.readAllLines()`**, применяется кодировка UTF-8.

2.2. Чтение и запись двоичных данных

Текстовый формат удобен для тестирования и отладки прикладного кода, поскольку он удобочитаем. Но он не столь эффективен для передачи данных, как двоичный формат. Поэтому в последующих разделах поясняется, каким образом организует ввод и вывод двоичных данных.

2.2.1. Интерфейсы `DataInput` и `DataOutput`

В интерфейсе `DataOutput` определяются следующие методы для записи чисел, символов, логических значений типа `boolean` или символьных строк в двоичном формате:

```
writeChars()      writeFloat()
writeByte()       writeDouble()
writeInt()        writeChar()
writeShort()      writeBoolean()
writeLong()       writeUTF()
```

Например, метод `writeInt()` всегда записывает целочисленное значение в виде 4-байтовой двоичной величины независимо от количества цифр, а метод `writeDouble()` — числовое значение с плавающей точкой типа `double` в виде 8-байтовой двоичной величины. Выводимый в итоге результат неудобочитаем, но в то же время объем требующегося пространства будет одинаковым для каждого значения заданного типа, а обратное считывание таких значений будет осуществляться намного быстрее, чем синтаксический анализ текста.

НА ЗАМЕТКУ! Для сохранения целочисленных значений и числовых значений с плавающей точкой имеются два разных способа, зависящих от используемой платформы. Допустим, имеется некоторое 4-байтовое значение типа **`int`**, скажем, десятичное число **`1234`**, или **`4D2`**

в шестнадцатеричном представлении (**1234 = 4 × 256 + 13 × 16 + 2**). С одной стороны, оно может быть сохранено таким образом, чтобы чтые первых байта в памяти занимал его самый старший байт: **00 00 04 D2**. Такой способ называется сохранением в формате с обратным порядком следования байтов от старшего к младшему. А с другой стороны, оно может быть сохранено таким образом, чтобы первым следовал самый младший байт: **D2 04 00 00**. Такой способ называется сохранением в формате с прямым порядком следования байтов от младшего к старшему. На платформах SPARC, например, применяется формат с обратным порядком следования байтов, а на платформах Pentium — формат с прямым порядком следования байтов.

Это может стать причиной серьезных осложнений. Например, данные сохраняются в исходном файле программы на C или C++ именно так, как это делает процессор. Вследствие этого перемещение даже простейших файлов данных с одной платформы на другую превращается в совсем не простую задачу. А в Java все значения записываются в формате с обратным порядком следования байтов от старшего к младшему независимо от типа процессора, что, соответственно, делает файлы данных в Java независящими от используемой платформы.

Метод `writeUTF()` записывает строковые данные, используя "модифицированную" версию 8-разрядного формата преобразования Юникода. Вместо стандартной кодировки UTF-8 символьные строки сначала представляются в кодировке UTF-16, а полученный результат кодируется по правилам UTF-8. Для символов с кодом больше `0xFFFF` модифицированная кодировка выглядит по-другому. Она применяется для обеспечения обратной совместимости с виртуальными машинами, которые были созданы в те времена, когда Юникод еще не мог выходить за рамки 16 битов.

Но такой "модифицированной" версией UTF-8 уже никто не пользуется, и поэтому для записи символьных строк, предназначенных для виртуальной машины Java, например, при создании программы, генерирующей байт-коды, следует использовать только метод `writeUTF()`, а для всех остальных целей — метод `writeChars()`.

Для обратного чтения данных можно воспользоваться следующими методами, определенными в интерфейсе `DataInput`:

```
readInt()        readDouble()
readShort()      readChar()
readLong()       readBoolean()
readFloat()      readUTF()
```

Интерфейс `DataInput()` реализуется в классе `DataInputStream`. Читать двоичные данные из файла можно простым сочетанием потока ввода типа `DataInputStream` с нужным источником байтов, например, типа `FileInputStream`, как показано ниже.

```
var in = new
    DataInputStream(new FileInputStream("employee.dat"));
```

Аналогично можно записывать двоичные данные с помощью класса `DataOutputStream`, реализующего интерфейс `DataOutput`, следующим образом:

```
var out = new
    DataOutputStream(new FileOutputStream("employee.dat"));
```

java.io.DataInput 1.0

- **boolean readBoolean()**
- **byte readByte()**
- **char readChar()**
- **double readDouble()**
- **float readFloat()**
- **int readInt()**
- **long readLong()**
- **short readShort()**

  Считывают значение заданного типа.
- **void readFully(byte[] *b*)**

  Считывает байты в массив ***b***, устанавливая блокировку до тех пор, пока не будут считаны все байты.
- **void readFully(byte[] *b*, int *off*, int *len*)**

  Считывает байты в массив ***b***, устанавливая блокировку до тех пор, пока не будут считаны все байты.
- **String readUTF()**

  Считывает символьную строку в "модифицированном" формате UTF-8.
- **int skipBytes(int *n*)**

  Пропускает ***n*** байтов, устанавливая блокировку до тех пор, пока не будут пропущены все необходимые байты.

java.io.DataOutput 1.0

- **void writeBoolean(boolean *b*)**
- **void writeByte(int *b*)**
- **void writeChar(int *c*)**
- **void writeDouble(double *d*)**
- **void writeFloat(float *f*)**
- **void writeInt(int *i*)**
- **void writeLong(long *l*)**
- **void writeShort(int *s*)**

  Записывают значение заданного типа.
- **void writeChars(String *s*)**

  Записывает все символы из строки.
- **void writeUTF(String *s*)**

  Записывает символьную строку в "модифицированном" формате UTF-8

2.2.2. Файлы с произвольным доступом

Класс RandomAccessFile позволяет отыскивать или записывать данные где угодно в файле. Для файлов на дисках всегда имеется возможность произвольного доступа, тогда как для потоков ввода-вывода данных через сетевой сокет такая возможность отсутствует. Файл с произвольным доступом может открываться только для чтения или же как для чтения, так и для записи. Требуемый режим доступа задается указанием во втором параметре конструктора символьной строки "r" (режим только для чтения) или символьной строки "rw" (режим для чтения и записи) соответственно, как показано ниже. Если существующий файл открывается как объект типа RandomAccessFile, он не удаляется.

```
var in = new RandomAccessFile("employee.dat", "r");
var inOut = new RandomAccessFile("employee.dat", "rw");
```

У любого файла с произвольным доступом имеется так называемый *указатель файла*, обозначающий позицию следующего байта, который будет считываться или записываться. Метод seek() устанавливает этот указатель на произвольную байтовую позицию в файле. Этому методу в качестве аргумента может быть передано целочисленное значение типа long в пределах от нуля до числового значения, обозначающего длину файла в байтах. А метод getFilePointer() возвращает текущую позицию указателя файла.

Класс RandomAccessFile реализует как интерфейс DataInput, так и интерфейс DataOutput. Для чтения данных и записи данных в файл с произвольным доступом применяются методы readInt() и writeInt(), а также методы readChar() и writeChar(), обсуждавшиеся в предыдущем разделе.

Рассмотрим в качестве примера программу, сохраняющую записи о работниках в файле с произвольным доступом. Все записи будут иметь одинаковые размеры. Это упрощает процесс чтения произвольной записи. Допустим, указатель файла требуется поместить на третью запись. Для этого достаточно установить указатель файла на соответствующей байтовой позиции и затем приступить к чтению нужной записи, как показано ниже.

```
long n = 3;
in.seek((n - 1) * RECORD_SIZE);
var e = new Employee();
e.readData(in);
```

Если же требуется сначала внести изменения в запись, а затем сохранить ее в том же самом месте, в таком случае нужно снова установить указатель файла на начало записи следующим образом:

```
in.seek((n - 1) * RECORD_SIZE);
e.writeData(out);
```

Для определения общего количества байтов в файле служит метод length(). Общее количество записей определяется путем деления длины файла на размер каждой записи:

```
long nbytes = in.length(); // длина файла в байтах
int nrecords = (int) (nbytes / RECORD_SIZE);
```

Целочисленные значения и числовые значения с плавающей точкой имеют фиксированный размер в двоичном формате, тогда как с символьными строками

дело обстоит немного сложнее. Для записи и чтения символьных строк фиксированного размера придется ввести два вспомогательных метода. В частности, приведенный ниже метод `writeFixedString()` записывает указанное количество кодовых единиц, отсчитывая от начала символьной строки. (Если кодовых единиц слишком мало, символьная строка дополняется нулевыми значениями.)

```
public static void writeFixedString(String s, int size,
                  DataOutput out) throws IOException
{
  for (int i = 0; i < size; i++)
  {
    char ch = 0;
    if (i < s.length()) ch = s.charAt(i);
    out.writeChar(ch);
  }
}
```

Приведенный ниже метод `readFixedString()` считывает символы из потока ввода (типа `InputStream`) до тех пор, пока не прочитает указанное в качестве параметра `size` количество кодовых единиц или пока не встретится символ с нулевым значением. В последнем случае все остальные нулевые значения в поле ввода пропускаются. Для повышения эффективности чтения символьных строк в этом методе используется класс `StringBuilder`.

```
public static String readFixedString(int size,
              DataInput in) throws IOException
{
  var b = new StringBuilder(size);
  int i = 0;
  var done = false;
  while (!done && i < size)
  {
    char ch = in.readChar();
    i++;
    if (ch == 0) done = true;
    else b.append(ch);
  }
  in.skipBytes(2 * (size - i));
  return b.toString();
}
```

Методы `writeFixedString()` и `readFixedString()` введены во вспомогательный класс `DataIO`. Для сохранения записи фиксированного размера все поля записываются в двоичном формате следующим образом:

```
DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
out.writeDouble(e.getSalary());
LocalDate hireDay = e.getHireDay();
out.writeInt(hireDay.getYear());
out.writeInt(hireDay.getMonthValue());
out.writeInt(hireDay.getDayOfMonth());
```

Обратное чтение данных выполняется так же просто:

```
String name = DataIO.readFixedString(
                    Employee.NAME_SIZE, in);
double salary = in.readDouble();
```

```
int y = in.readInt();
int m = in.readInt();
int d = in.readInt();
```

Подсчитаем размер каждой записи. Для строк с Ф.И.О. выделим по 40 символов. Следовательно, каждая запись будет содержать по 100 байтов:

- по 40 символов, т.е. по 80 байтов, на каждые Ф.И.О.;
- по 1 числовому значению типа `double`, т.е. по 8 байтов, на размер зарплаты;
- по 3 числовых значения типа **`int`**, т.е. по 12 байтов, на дату зачисления на работу.

Программа из листинга 2.2 делает три записи в файле данных, а затем читает их оттуда в обратном порядке. Для эффективного выполнения данного процесса требуется произвольный доступ к файлу, а в данном случае — доступ сначала к последней записи.

Листинг 2.2. Исходный код из файла **`randomAccess/RandomAccessTest.java`**

```
1  package randomAccess;
2
3  import java.io.*;
4  import java.time.*;
5
6  /**
7   * @version 1.14 2018-05-01
8   * @author Cay Horstmann
9   */
10 public class RandomAccessTest
11 {
12   public static void main(String[] args)
13         throws IOException
14   {
15     var staff = new Employee[3];
16
17     staff[0] = new Employee("Carl Cracker", 75000,
18                             1987, 12, 15);
19     staff[1] = new Employee("Harry Hacker", 50000,
20                             1989, 10, 1);
21     staff[2] = new Employee("Tony Tester", 40000,
22                             1990, 3, 15);
23
24     try (var out = new DataOutputStream(
25          new FileOutputStream("employee.dat")))
26     {
27       // сохранить все записи о работниках
28       // в файле employee.dat
29       for (Employee e : staff)
30         writeData(out, e);
31     }
32
33     try (var in = new RandomAccessFile(
34          "employee.dat", "r"))
```

```
35      {
36        // извлечь все записи в новый массив
37
38        // определить размер массива
39        int n = (int)(in.length() / Employee.RECORD_SIZE);
40        var newStaff = new Employee[n];
41
42        // прочитать записи о работниках
43        // в обратном порядке
44        for (int i = n - 1; i >= 0; i--)
45        {
46          newStaff[i] = new Employee();
47          in.seek(i * Employee.RECORD_SIZE);
48          newStaff[i] = readData(in);
49        }
50
51        // вывести вновь прочитанные записи о работниках
52        for (Employee e : newStaff)
53          System.out.println(e);
54      }
55    }
56
57    /**
58     * Записывает сведения о работниках
59     * в поток вывода данных
60     * @param out Поток вывода данных
61     * @param e Работник
62     */
63    public static void writeData(DataOutput out,
64                  Employee e) throws IOException
65    {
66      DataIO.writeFixedString(e.getName(),
67                              Employee.NAME_SIZE, out);
68      out.writeDouble(e.getSalary());
69
70      LocalDate hireDay = e.getHireDay();
71      out.writeInt(hireDay.getYear());
72      out.writeInt(hireDay.getMonthValue());
73      out.writeInt(hireDay.getDayOfMonth());
74    }
75
76    /**
77     * Читает сведения о работниках
78     * из потока ввода данных
79     * @param in Поток ввода данных
80     * @return Возвращает работника
81     */
82    public static Employee readData(DataInput in)
83          throws IOException
84    {
85      String name = DataIO.readFixedString(
86                    Employee.NAME_SIZE, in);
87      double salary = in.readDouble();
88      int y = in.readInt();
```

```
89      int m = in.readInt();
90      int d = in.readInt();
91      return new Employee(name, salary, y, m - 1, d);
92    }
93 }
```

java.io.RandomAccessFile 1.0

- **RandomAccessFile(String *file*, String *mode*)**
- **RandomAccessFile(File *file*, String *mode*)**

  Открывают заданный файл для произвольного доступа. Строковое значение **"r"** параметра ***mode*** обозначает режим только для чтения, строковое значение **"rw"** — режим для чтения и записи, строковое значение **"rws"** — режим для чтения и синхронной записи данных на диск вместе с метаданными при каждом обновлении файла, а строковое значение **"rwd"** — режим чтения и синхронной записи на диск только данных.
- **long getFilePointer()**

  Возвращает сведения о текущем местоположении указателя файла.
- **void seek(long *pos*)**

  Устанавливает указатель файла на позицию ***pos*** от начала файла.
- **long length()**

  Возвращает длину файла в байтах.

2.2.3. ZIP-архивы

В ZIP-архивах можно хранить один или несколько файлов (как правило) в уплотненном формате. У каждого ZIP-архива имеется заголовок, содержащий сведения вроде имени файла или применявшегося для него алгоритма сжатия. В языке Java для чтения ZIP-архивов служит класс `ZipInputStream`. В каждом таком архиве всегда требуется просматривать отдельные *записи*. Метод `getNextEntry()` возвращает описывающий запись объект типа `ZipEntry`. Чтобы получить поток ввода для чтения записи из архива, эту запись следует передать методу `getInputStream()`. Далее вызывается метод `closeEntry()` для перехода к чтению следующей записи. Ниже приведен типичный фрагмент кода для чтения содержимого архивного ZIP-файла.

```
var zin = new
    ZipInputStream(new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
  InputStream in = zin.getInputStream(entry);
  прочитать содержимое потока ввода in
  zin.closeEntry();
}
zin.close();
```

Для записи в ZIP-архив служит класс `ZipOutputStream`. В этом случае для каждой записи, которую требуется ввести в ZIP-архив, создается объект типа `ZipEntry`. Требуемое имя файла передается конструктору класса `ZipEntry`, где

устанавливаются остальные параметры вроде даты создания файла и алгоритма распаковки. По желанию эти параметры могут быть переопределены. Далее вызывается метод `putNextEntry()` из класса `ZipOutputStream`, чтобы начать процесс записи нового файла. С этой целью данные из самого файла направляются в поток вывода в ZIP-архив, а по завершении вызывается метод `closeEntry()`. Затем описанные здесь действия выполняются повторно для всех остальных файлов, которые требуется сохранить в ZIP-архиве. Ниже приведена общая структура кода, требующегося для этой цели.

```
var fout = new FileOutputStream("test.zip");
var zout = new ZipOutputStream(fout);
для всех файлов
{
  var ze = new ZipEntry(имя_файла);
  zout.putNextEntry(ze);
  направить данные в поток вывода zout
  zout.closeEntry();
}
zout.close();
```

НА ЗАМЕТКУ! Архивные JAR-файлы (см. главу 4 первого тома настоящего издания) представляют собой те же самые ZIP-файлы, но только они содержат записи несколько иного вида, называемые манифестами. Для чтения и записи манифестов служат классы **`JarInputStream`** и **`JarOutputStream`**.

Потоки ввода из ZIP-архивов служат отличным примером, раскрывающим истинный потенциал потоковой абстракции. При чтении данных, хранящихся в уплотненном виде, не нужно особенно беспокоиться о том, будут ли они разуплотняться по мере запрашивания. Более того, источником байтов в потоках ввода из ZIP-архивов совсем не обязательно должен быть именно файл: данные, уплотненные в формате ZIP, могут поступать и через сетевое соединение.

НА ЗАМЕТКУ! В разделе 2.4.8 поясняется, как осуществить доступ к ZIP-архиву без специального прикладного интерфейса API, используя класс **`FileSystem`**, внедренный в версии Java 7.

`java.util.zip.ZipInputStream` 1.1

- **`ZipInputStream(InputStream in)`**
  Создает объект типа **`ZipInputStream`**, позволяющий распаковывать данные из указанного объекта типа **`InputStream`**.
- **`ZipEntry getNextEntry()`**
  Возвращает объект типа **`ZipEntry`** для следующей записи или пустое значение **`null`**, если записей больше нет.
- **`void closeEntry()`**
  Закрывает текущую открытую запись в ZIP-архиве. Далее может быть прочитана следующая запись с помощью метода **`getNextEntry()`**.

java.util.zip.ZipOutputStream 1.1

- **ZipOutputStream(OutputStream *out*)**
  Создает объект типа **ZipOutputStream**, позволяющий записывать уплотненные данные в указанный поток вывода типа **OutputStream**.
- **void putNextEntry(ZipEntry *ze*)**
  Записывает данные из указанной записи типа **ZipEntry** в поток вывода и устанавливает его в положение для вывода следующей порции данных. После этого данные могут быть направлены в этот поток вывода с помощью метода **write()**.
- **void closeEntry()**
  Закрывает текущую открытую запись в ZIP-архиве. Для перехода к следующей записи используется метод **putNextEntry()**.
- **void setLevel(int *level*)**
  Устанавливает степень сжатия для последующих записей. По умолчанию устанавливается значение степени сжатия **Deflater.DEFAULT_COMPRESSION**. Генерирует исключение типа **IllegalArgumentException**, если заданная степень сжатия недействительна.
- **void setMethod(int *method*)**
  Устанавливает алгоритм сжатия по умолчанию для любых записей, направляемых в поток вывода типа **ZipOutputStream**, без указания конкретного алгоритма сжатия данных. В качестве параметра ***method*** можно указать значение **DEFLATED** или **STORED**, обозначающее конкретный алгоритм сжатия.

java.util.zip.ZipEntry 1.1

- **ZipEntry(String *name*)**
  Создает запись в Zip-архиве с заданным именем.
- **long getCrc()**
  Возвращает значение контрольной суммы CRC32 для данной записи типа **ZipEntry**.
- **String getName()**
  Возвращает имя данной записи.
- **long getSize()**
  Возвращает размер данной записи без сжатия или значение **-1**, если размер записи без уплотнения неизвестен.
- **boolean isDirectory()**
  Возвращает логическое значение **true**, если данная запись является каталогом.
- **void setMethod(int *method*)**
  Задает алгоритм сжатия записей.
- **void setSize(long *size*)**
  Устанавливает размер данной записи. Требуется только в том случае, если задан алгоритм сжатия данных **STORED**.
- **void setCrc(long *crc*)**
  Устанавливает контрольную сумму CRC32 для данной записи. Для вычисления этой суммы должен использоваться класс **CRC32**. Требуется только в том случае, если задан алгоритм сжатия данных **STORED**.

java.util.zip.ZipFile 1.1

- **ZipFile(String *name*)**
- **ZipFile(File *file*)**

  Создают объект типа **ZipFile** для чтения из заданной символьной строки или объекта типа **File**.
- **Enumeration entries()**

  Возвращает объект типа **Enumeration**, перечисляющий объекты типа **ZipEntry**, описывающие записи из архива типа **ZipFile**.
- **ZipEntry getEntry(String *name*)**

  Возвращает запись, соответствующую указанному имени, или пустое значение **null**, если такой записи не существует.
- **InputStream getInputStream(ZipEntry *ze*)**

  Возвращает поток ввода типа **InputStream** для указанной записи.
- **String getName()**

  Возвращает путь к данному ZIP-архиву.

2.3. Потоки ввода-вывода и сериализация объектов

Пользоваться форматом записей фиксированной длины, безусловно, удобно, если сохранять объекты одинакового типа. Но ведь объекты, создаваемые в объектно-ориентированных программах, редко бывают одного и того же типа. Например, может существовать массив `staff`, номинально представляющий собой массив записей типа `Employee`, но фактически содержащий объекты, которые, по существу, являются экземплярами какого-нибудь подкласса вроде `Manager`.

Конечно, можно было бы подобрать такой формат данных, который позволял бы сохранять подобные полиморфные коллекции, но, к счастью, в этом нет никакой необходимости. В языке Java поддерживается универсальный механизм, называемый *сериализацией объектов* и предоставляющий возможность записать любой объект в поток ввода-вывода, а в дальнейшем прочитать его снова. (Происхождение термина *сериализация* подробно объясняется далее в этой главе.)

2.3.1. Сохранение и загрузка сериализируемых объектов

Для сохранения данных объектов необходимо прежде всего открыть поток вывода объектов типа `ObjectOutputStream` следующим образом:

```
var out = new ObjectOutputStream(
          new FileOutputStream("employee.dat"));
```

Далее для сохранения объекта остается лишь вызвать метод `writeObject()` из класса `ObjectOutputStream`, как показано в приведенном ниже фрагменте кода.

```
var harry = new Employee("Harry Hacker", 50000,
                         1989, 10, 1);
var boss = new Manager("Carl Cracker", 80000,
                       1987, 12, 15);
```

```
out.writeObject(harry);
out.writeObject(boss);
```

А для того чтобы прочитать данные объектов обратно, необходимо получить сначала объект типа `ObjectInputStream`, т.е. поток ввода объектов, следующим образом:

```
var in = new ObjectInputStream(
         new FileInputStream("employee.dat"));
```

И затем извлечь объекты в том порядке, в каком они записывались, вызвав метод `readObject()`, как показано ниже.

```
var e1 = (Employee) in.readObject();
var e2 = (Employee) in.readObject();
```

Имеется, однако, одно изменение, которое нужно внести в любой класс, объекты которого требуется сохранить и восстановить в потоке ввода-вывода объектов, а именно: каждый такой класс должен обязательно реализовать интерфейс `Serializable` следующим образом:

```
class Employee implements Serializable { . . . }
```

У интерфейса `Serializable` отсутствуют методы, поэтому изменять каким-то образом свои собственные классы не нужно. В этом отношении интерфейс `Serializable` подобен интерфейсу `Cloneable`, рассматривавшемуся в главе 6 первого тома настоящего издания. Но для того чтобы сделать класс пригодным для клонирования, все равно требовалось переопределить метод `clone()` из класса `Object`. А для того чтобы сделать класс пригодным для сериализации, ничего больше делать не нужно.

НА ЗАМЕТКУ! Записывать и читать только объекты можно и с помощью методов **`writeObject()`** и **`readObject()`**. Что же касается значений простых типов, то для их ввода-вывода следует применять такие методы, как **`writeInt()`** и **`readInt()`** или **`writeDouble()`** и **`readDouble()`**. (Классы потоков ввода-вывода объектов реализуют интерфейсы **`DataInput`** и **`DataOutput`**.)

Класс `ObjectOutputStream` просматривает подспудно все поля объектов и сохраняет их содержимое. Так, при записи объекта типа `Employee` в поток вывода записывается содержимое полей Ф.И.О., даты зачисления на работу и зарплаты работника.

Необходимо, однако, рассмотреть очень важный вопрос: что произойдет, если один объект совместно используется рядом других объектов? Чтобы проиллюстрировать важность данного вопроса, внесем одно небольшое изменение в класс `Manager`. В частности, допустим, что у каждого руководителя имеется свой секретарь, как показано в приведенном ниже фрагменте кода.

```
class Manager extends Employee
{
  private Employee secretary;
  . . .
}
```

Теперь каждый объект типа `Manager` будет содержать ссылку на объект типа `Employee`, описывающий секретаря. Безусловно, два руководителя вполне могут пользоваться услугами одного и того же секретаря, как показано на рис. 2.5 и в следующем фрагменте кода:

```
var harry = new Employee("Harry Hacker", . . .);
var carl = new Manager("Carl Cracker", . . .);
carl.setSecretary(harry);
var tony = new Manager("Tony Tester", . . .);
tony.setSecretary(harry);
```

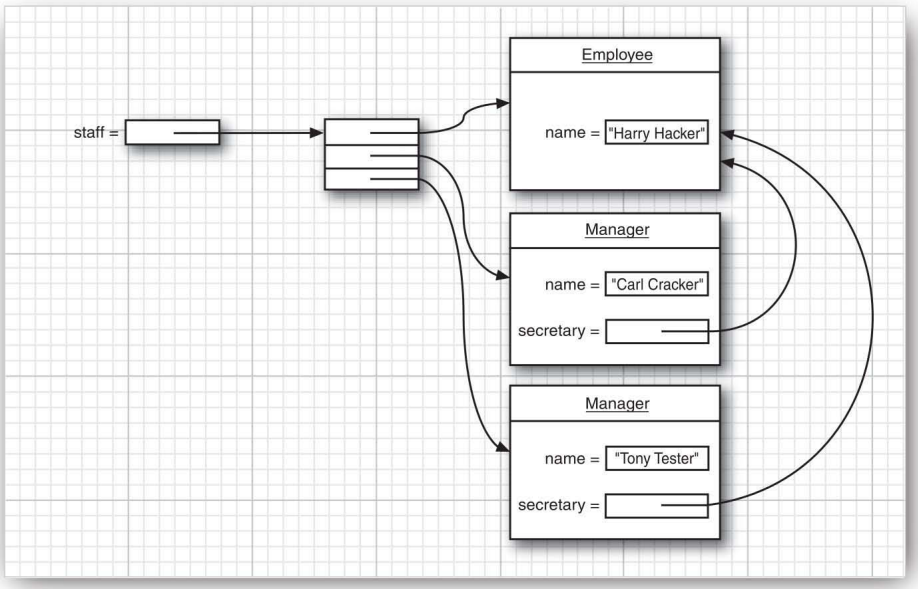


Рис. 2.5. Два руководителя могут совместно пользоваться услугами одного и того же работника в качестве секретаря

Сохранение такой разветвленной сети объектов оказывается непростой задачей. Разумеется, сохранять и восстанавливать адреса ячеек памяти для объектов секретарей нельзя. При повторной загрузке каждый такой объект, скорее всего, будет занимать уже совершенно другую ячейку памяти, а не ту, которую он занимал первоначально.

Поэтому каждый такой объект сохраняется под *серийным номером*, откуда, собственно говоря, и происходит название механизма *сериализации объектов*. Ниже описывается порядок действий при сериализации объектов.

1. Серийный (т.е. порядковый) номер связывается с каждой встречающейся ссылкой на объект, как показано на рис. 2.6.
2. Если ссылка на объект встречается впервые, данные из этого объекта сохраняются в потоке ввода-вывода объектов.

3. Если же данные были ранее сохранены, просто добавляется метка "`same as previously saved object with serial number x`" (совпадает с объектом, сохраненным ранее под серийным номером x).

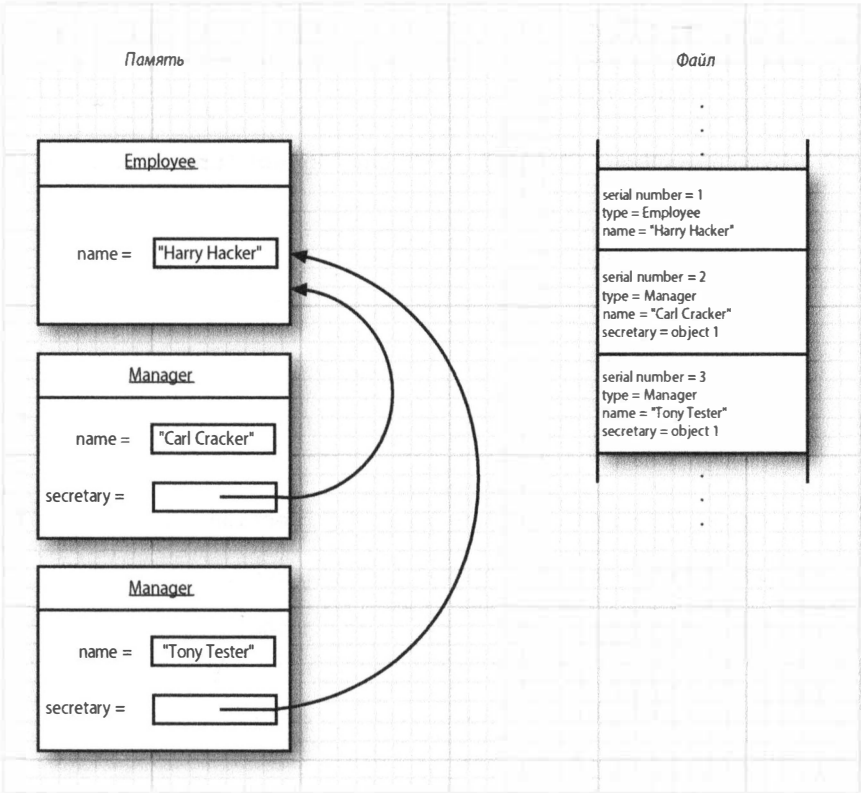


Рис. 2.6. Пример сериализации объектов

При чтении объектов обратно из потока их ввода-вывода порядок действий меняется на обратный.

1. Если объект впервые указывается в потоке ввода-вывода объектов, он создается и инициализируется данными из потока, а связь серийного номера со ссылкой на объект запоминается.
2. Если встречается метка "`same as previously saved object with serial number x`", то извлекается ссылка на объект по данному серийному номеру.

НА ЗАМЕТКУ! В этой главе демонстрируется, каким образом механизм сериализации можно применять для сохранения объектов в файле на диске и извлечения их в точном соответствии с тем, как они сохранялись. Другой очень важной областью применения данного механизма является передача коллекции объектов по сетевому соединению на другой компьютер. Исходные адреса ячеек памяти не имеют никакого значения при взаимодействии с другим процессором, как и при обращении к файлу. Заменяя адреса ячеек памяти серийными номерами, механизм сериализации делает вполне возможным перенос коллекций объектов с одной машины на другую.

В листинге 2.3 приведен исходный код примера программы, способной сохранять и перезагружать сеть объектов типа `Employee` и `Manager` (некоторые из руководителей пользуются услугами общего работника в качестве секретаря). Обратите внимание на то, что объект секретаря остается однозначным после повторной перезагрузки, т.е. когда работник, представленный объектом, хранящимся в элементе массива `newStaff[1]`, получает повышение, это отражается в полях `secretary` объектов типа `Manager`.

Листинг 2.3. Исходный код из файла **objectStream/ObjectStreamTest.java**

```
1  package objectStream;
2
3  import java.io.*;
4
5  /**
6   * @version 1.11 2018-05-01
7   * @author Cay Horstmann
8   */
9  class ObjectStreamTest
10 {
11   public static void main(String[] args)
12         throws IOException, ClassNotFoundException
13   {
14     var harry = new Employee("Harry Hacker", 50000,
15                              1989, 10, 1);
16     var carl = new Manager("Carl Cracker", 80000,
17                            1987, 12, 15);
18     carl.setSecretary(harry);
19     var tony = new Manager("Tony Tester", 40000,
20                            1990, 3, 15);
21     tony.setSecretary(harry);
22
23     var staff = new Employee[3];
24
25     staff[0] = carl;
26     staff[1] = harry;
27     staff[2] = tony;
28
29     // сохранить записи обо всех работниках
30     // в файле employee.dat
31     try (var out = new ObjectOutputStream(
32          new FileOutputStream("employee.dat")))
33     {
34       out.writeObject(staff);
35     }
36
37     try (var in = new ObjectInputStream(
38          new FileInputStream("employee.dat")))
39     {
40       // извлечь все записи в новый массив
41
42       var newStaff = (Employee[]) in.readObject();
```

```
43
44        // поднять зарплату секретарю
45        newStaff[1].raiseSalary(10);
46
47        // вывести вновь прочитанные записи работников
48        for (Employee e : newStaff)
49          System.out.println(e);
50      }
51    }
52 }
```

java.io.ObjectOutputStream 1.1

- **ObjectOutputStream(OutputStream *out*)**

  Создает поток вывода объектов типа **ObjectOutputStream**, чтобы объекты можно было записывать в указанный поток вывода типа **OutputStream**.

- **void writeObject(Object *obj*)**

  Записывает указанный объект в поток вывода объектов типа **ObjectOutputStream**. Сохраняет класс объекта, его сигнатуру и значения из любого нестатического и непереходного поля данного класса и его суперклассов.

java.io.ObjectInputStream 1.1

- **ObjectInputStream(InputStream *in*)**

  Создает поток ввода объектов типа **ObjectInputStream** для обратного чтения данных об объектах из указанного потока ввода типа **InputStream**.

- **Object readObject()**

  Читает объект из потока ввода объектов типа **ObjectInputStream**. В частности, читает обратно класс объекта, его сигнатуры, а также значения всех непереходных и нестатических полей этого класса и всех его суперклассов. Осуществляет десериализацию для восстановления многих ссылок на объекты.

2.3.2. Представление о формате файлов для сериализации объектов

При сериализации объектов их данные сохраняются в файле определенного формата. Безусловно, методами `writeObject()` и `readObject()` можно было бы воспользоваться, даже не зная, каким образом выглядит последовательность байтов, представляющая объекты в файле. Тем не менее изучение формата данных очень полезно для получения ясного представления о процессе потоковой обработки объектов. Рассматриваемый здесь материал носит до некоторой степени технический характер, поэтому, если вас не интересуют подробности реализации, можете пропустить этот раздел.

Каждый файл начинается с состоящего из двух байтов "магического числа" `AC ED`, сопровождаемого номером версии формата сериализации объектов, который в настоящее время имеет вид `00 05`. (Здесь и далее в этом разделе байты представлены шестнадцатеричными числами.) Далее в файле находится последовательность

объектов, которые следуют друг за другом именно в том порядке, в каком они сохранялись. Строковые объекты сохраняются в следующем виде:

```
74      Двухбайтовое число, обозначающее длину файла      Символы
```

Например, символьная строка `"Harry"` сохраняется в файле следующим образом:

```
74 00 05 Harry
```

Символы Юникода из строковых объектов сохраняются в "модифицированном" формате UTF-8. Вместе с объектом должен непременно сохраняться и его класс. Описание класса включает в себя следующее.

- Имя класса.
- *Однозначный идентификатор порядкового номера версии*, представляющий собой отпечаток типов полей данных и сигнатур методов.
- Набор флагов, описывающих метод сериализации.
- Описание полей данных.

Для получения отпечатка сначала каноническим способом упорядочиваются описания класса, суперкласса, интерфейсов, типов полей и сигнатур методов, а затем к этим данным применяется алгоритм так называемого безопасного хеширования (Secure Hash Algorithm — SHA).

Алгоритм SHA позволяет быстро получить "отпечаток" с большого блока данных. Этот "отпечаток" всегда представляет собой 20-байтовый пакет данных, каким бы ни был размер исходных данных. Он создается в результате выполнения над данными некоторой искусно составленной последовательности поразрядных операций, что дает практически полную уверенность, что при любом изменении данных изменится и сам "отпечаток". (Подробнее с алгоритмом SHA можно ознакомиться в книге *Cryptography and Network Security: Principles and Practice, Seventh Edition* Уильяма Столингса (William Stallings; издательство Prentice Hall, 2016 г.) Но в механизме сериализации для "отпечатка" класса используются только первые 8 байтов кода SHA. И тем не менее это гарантирует, что при изменении полей данных или методов изменится и "отпечаток" класса.

При чтении объекта его "отпечаток" сравнивается с текущим "отпечатком" класса. Если они не совпадают, это означает, что после записи объекта определение класса изменилось, а следовательно, генерируется исключение. На практике классы постепенно усовершенствуются, и поэтому от прикладной программы, возможно, потребуется способность читать прежние версии объектов. Более подробно данный вопрос обсуждается в разделе 2.3.5.

Идентификатор класса сохраняется в следующей последовательности.

- `72`
- Двухбайтовое число, обозначающее длину имени класса
- Имя класса
- 8-байтовый "отпечаток"
- Однобайтовый флаг
- Двухбайтовое число, обозначающее количество дескрипторов полей данных

- Дескрипторы полей данных
- 78 (конечный маркер)
- Тип суперкласса (если таковой отсутствует, то 70)

Байт флага состоит из трех битовых масок, определяемых в классе `java.io.ObjectStreamConstants` следующим образом:

```
static final byte SC_WRITE_METHOD = 1;
    // в данном классе имеется метод writeObject(),
    // записывающий дополнительные данные
static final byte SC_SERIALIZABLE = 2;
    // в данном классе классе реализуется
    // интерфейс Serializable
static final byte SC_EXTERNALIZABLE = 4;
    // в данном классе реализуется
    // интерфейс Externalizable
```

Более подробно интерфейс `Externalizable` обсуждается далее в этой главе, а до тех пор достаточно сказать, что в классах, реализующих интерфейс `Externalizable`, предоставляются специальные методы чтения и записи, которые принимают данные, выводимые из полей экземпляров этих классов. Рассматриваемые здесь классы реализуют интерфейс `Serializable` и имеют значение флага 02. Класс `java.util.Date` также реализует интерфейс `Serializable`, но помимо этого он определяет свои методы `readObject()` и `writeObject()` и имеет значение флага 03.

Каждый дескриптор поля данных имеет следующий формат.

- Однобайтовый код типа
- Двухбайтовое число, обозначающее длину имени поля
- Имя поля
- Имя класса (если поле является объектом)

Код типа может принимать одно из следующих значений.

B	**byte**
C	**char**
D	**double**
F	**float**
I	**int**
J	**long**
L	объект
S	**short**
Z	**boolean**
[	массив

Если код типа принимает значение `L`, после имени поля следует тип поля. Символьные строки имен классов и полей *не* начинаются со строкового кода 74, тогда как символьные строки типов полей начинаются именно с него. Для имен типов полей используется несколько иная кодировка, а именно: формат, применяемый в платформенно-ориентированных методах. Например, поле зарплаты из класса `Employee` кодируется следующим образом:

```
D 00 06 salary
```

В качестве примера ниже полностью показан дескриптор класса `Employee`.

```
72    00 08 Employee
      E6 D2 86 7D AE AC 18 1B 02  Отпечаток и флаги
      00 03                       Количество полей экземпляра
      D 00 06 salary              Тип и имя поля экземпляра
      L 00 07 hireDay             Тип и имя поля экземпляра
      74 00 10 Ljava/util/Date;   Имя класса для поля экземпляра: Date
      L 00 04 name                Тип и имя поля экземпляра
      74 00 12 Ljava/lang/String; Имя класса для поля экземпляра: String
      78                          Конечный маркер
      70                          Суперкласс отсутствует
```

Эти дескрипторы получаются довольно длинными. Поэтому если дескриптор одного и того же класса снова потребуется в файле, то для этой цели используется следующая сокращенная форма:

```
71    4-байтовый порядковый номер
```

Порядковый (иначе называемый серийным) номер обозначает упоминавшийся выше явный дескриптор класса. А порядок нумерации рассматривается далее. Объект сохраняется в следующем формате:

```
73    Дескриптор класса    Данные объекта
```

В качестве примера ниже показано, каким образом объект типа `Employee` сохраняется в файле.

```
40 E8 6A 00 00 00 00 00          Значение поля salary: double
73                               Значение поля hireDay: новый объект
   71 00 7E 00 08                Существующий класс java.util.Date
   77 08 00 00 00 91 1B 4E B1 80 78  Внешнее хранилище (рассматривается ниже)
74 00 0C Harry Hacker            Значение поля name: String
```

Как видите, в файле данных сохраняется достаточно сведений для восстановления объекта типа `Employee`. А массивы сохраняются в следующем формате.

```
75    Дескриптор класса    4-байтовое число, обозначающее общее количество записей    Записи
```

Имя класса массива в дескрипторе класса указывается в том же формате, что и в платформенно-ориентированных методах (т.е. немного иначе, чем в формате, используемом для имен классов в других дескрипторах классов). В этом формате имена классов начинаются с буквы `L`, а завершаются точкой с запятой. Например, массив из трех объектов типа `Employee` будет начинаться следующим образом.

```
75                               Массив
   72    00 0B [LEmployee;       Новый класс, длина строки, имя класса Employee[]
         00 00                   Количество полей экземпляра
         78                      Конечный маркер
         70                      Суперкласс отсутствует
         00 00 00 03             Количество записей в массиве
```

Обратите внимание на то, что отпечаток массива объектов типа `Employee` отличается от отпечатка самого класса `Employee`. При сохранении в выходном файле всем объектам (массивам и символьным строкам включительно), а также всем дескрипторам классов присваиваются порядковые номера. Эти номера начинаются с кодовой последовательности `00 7E 00 00`.

Как упоминалось ранее, дескриптор любого конкретного класса указывается полностью в файле только один раз, а все последующие дескрипторы ссылаются на него. Так, в предыдущем примере повторяющаяся ссылка на класс `Date` кодировалась следующим образом: `71 00 7E 00 08`.

Тот же самый механизм применяется и для объектов. Так, если записывается ссылка на сохраненный ранее объект, она сохраняется точно так же, т.е. в виде маркера `71`, после которого следует порядковый номер. Указывает ли ссылка на объект или же на дескриптор класса, всегда можно выяснить из контекста. И, наконец, пустая ссылка сохраняется следующим образом: `70`.

Ниже приведен снабженный комментариями результат вывода из программы `ObjectStreamTest`, представленной в листинге 2.3. По желанию вы можете запустить эту программу на выполнение, посмотреть результат вывода данных в шестнадцатеричном виде из памяти в файл `employee.dat` и сравнить его с приведенным ниже результатом. Наиболее важные строки находятся ближе к концу и демонстрируют ссылку на сохраненный ранее объект.

```
AC ED 00 05                                     Заголовок файла
75                                              Массив staff (порядковый номер #1)
   72 00 0B [LEmployee;                         Новый класс, длина строки, имя класса
                                                Employee[] (порядковый номер #0)
      FC BF 36 11 C5 91 11 C7 02                "Отпечаток" и флаги
      00 00                                     Количество полей экземпляра
      78                                        Конечный маркер
      70                                        Суперкласс отсутствует
      00 00 00 03                               Количество записей в массиве
73                                              staff[0] — новый объект (порядковый номер
                                                #7)
   72 00 07 Manager                             Новый класс, длина строки, имя класса
                                                (порядковый номер #2)
      36 06 AE 13 63 8F 59 B7 02                Отпечаток и флаги
      00 01                                     Количество полей данных
      L  00 09 secretary                        Тип и имя поля экземпляра
      74 00 0A LEmployee;                       Имя класса для поля экземпляра — String
                                                (порядковый номер #3)
      78                                        Конечный маркер
      72 00 08 Employee                         Суперкласс — новый класс, длина строки, имя
                                                класса (порядковый номер #4)
         E6 D2 86 7D AE AC 18 1B 02             Отпечаток и флаги
         00 03                                  Количество полей экземпляра
         D 00 06 salary                         Тип и имя поля экземпляра
         L 00 07 hireDay                        Тип и имя поля экземпляра
         74 00 10 Ljava/util/Date;              Имя класса для поля экземпляра — String
                                                (порядковый номер #5)
         L 00 04 name                           Тип и имя поля экземпляра
         74 00 12 Ljava/lang/String;            Имя класса для поля экземпляра — String
                                                (порядковый номер #6)
         78                                     Конечный маркер
         70                                     Суперкласс отсутствует
40 F3 88 00 00 00 00 00                         Значение поля salary: double
```

`73`	Значение поля **secretary**: новый объект (порядковый номер **#9**)
`72 00 0E java.util.Date`	Новый класс, длина строки, имя класса (порядковый номер **#8**)
`68 6A 81 01 4B 59 74 19 03`	Отпечаток и флаги
`00 00`	Переменные экземпляра отсутствуют
`78`	Конечный маркер
`77 08`	Внешнее хранилище, количество байтов
`00 00 00 83 E9 39 E0 00`	Дата
`78`	Конечный маркер
`74 00 0C Carl Cracker`	Значение поля **name**: **String** (порядковый номер **#10**)
`73`	Значение поля **secretary**: новый объект (порядковый номер **#11**)
`71 00 7E 00 04`	Существующий класс (использовать порядковый номер **#4**)
`40 E8 6A 00 00 00 00 00`	Значение поля **salary**: **double**
`73`	Значение поля **hireDay**: новый объект (порядковый номер **#12**)
`71 00 7E 00 08`	Существующий класс (использовать порядковый номер **#8**)
`77 08`	Внешнее хранилище, количество байтов
`00 00 00 91 1B 4E B1 80`	Дата
`78`	Конечный маркер
`74 00 0C Harry Hacker`	Значение поля **name**: **String** (порядковый номер **#13**)
`71 00 7E 00 0B`	**staff[1]**: существующий объект (использовать порядковый номер **#11**)
`73`	**staff[2]**: новый объект (порядковый номер **#14**)
`71 00 7E 00 02`	Существующий класс (использовать порядковый номер **#2**)
`40 E3 88 00 00 00 00 00`	Значение поля **salary**: **double**
`73`	Значение поля **hireDay**: новый объект (порядковый номер **#15**)
`71 00 7E 00 08`	Существующий класс (использовать порядковый номер **#8**)
`77 08`	Внешнее хранилище, количество байтов
`00 00 00 94 6D 3E EC 00 00`	Дата
`78`	Конечный маркер
`74 00 0B Tony Tester`	Значение поля **name**: **String** (порядковый номер **#16**)
`71 00 7E 00 0B`	Значение поля **secretary**: существующий объект (использовать порядковый номер **#11**)

Разумеется, изучение этих кодов вряд ли увлекательнее чтения телефонного справочника. И хотя знать точный формат файла совсем не обязательно (если только изменение данных не преследует какую-нибудь злонамеренную цель), вам не помешает ясно представлять себе, что поток ввода-вывода содержит подробное описание всех направляемых в него объектов наряду с достаточными сведениями для восстановления как отдельных объектов, так и массивов объектов.

Самое главное — запомнить следующее.

- Поток ввода-вывода объектов содержит сведения о типах и полях данных всех входящих в него объектов.
- Для каждого объекта назначается порядковый (т.е. серийный) номер.
- Повторные вхождения того же самого объекта сохраняются в виде ссылок на его порядковый номер.

2.3.3. Видоизменение исходного механизма сериализации

Некоторые поля данных не должны вообще подвергаться сериализации, как, например, поля с целочисленными значениями, в которых хранятся дескрипторы файлов или дескрипторы окон, имеющие значение только для платформенно-ориентированных методов. Такие сведения, без сомнения, становятся бесполезными при последующей повторной загрузке объекта или при его переносе на другую машину. На самом деле неверные значения в таких полях могут даже привести к аварийному завершению платформенно-ориентированных методов. Поэтому в Java предусмотрен простой механизм, позволяющий предотвращать сериализацию подобных полей. Все, что для этого требуется, — объявить их как переходные с ключевым словом `transient`. Объявлять их подобным образом требуется и в том случае, если они не относятся к сериализируемым классам. А при сериализации объектов переходные поля всегда пропускаются.

Механизм сериализации предусматривает для отдельных классов возможность дополнять стандартный режим чтения и записи процедурами проверки правильности данных или любыми другими требующимися действиями. В сериализируемом классе могут быть определены методы с приведенными ниже сигнатурами. В таком случае поля данных больше не станут автоматически подвергаться сериализации, а вместо нее будут вызываться эти методы.

```
private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream out)
        throws IOException;
```

Рассмотрим типичный пример. В пакете `java.awt.geom` некоторые классы вроде `Point2D.Double` не являются сериализируемыми. Допустим, требуется сериализовать класс `LabeledPoint`, содержащий поля `String` и `Point2D.Double`. Для этого поле `Point2D.Double`, прежде всего, объявляется как переходное (`transient`), чтобы не возникло исключение типа `NotSerializableException`, как показано ниже.

```
public class LabeledPoint implements Serializable
{
  private String label;
  private transient Point2D.Double point;
  . . .
}
```

Далее в методе `writeObject()` сначала записывается дескриптор объекта и поле `label` типа `String`. Для этого служит специальный метод `defaultWriteObject()` из класса `ObjectOutputStream`, который может вызываться только из метода `writeObject()` сериализируемого класса. Затем

координаты точки записываются в поток вывода данных с помощью стандартных методов, вызываемых из класса `DataOutput` следующим образом:

```
private void writeObject(ObjectOutputStream out)
        throws IOException
{
  out.defaultWriteObject();
  out.writeDouble(point.getX());
  out.writeDouble(point.getY());
}
```

В методе `readObject()` выполняется обратный процесс:

```
private void readObject(ObjectInputStream in)
        throws IOException
{
  in.defaultReadObject();
  double x = in.readDouble();
  double y = in.readDouble();
  point = new Point2D.Double(x, y);
}
```

Еще одним примером может служить класс `java.util.Date`, предоставляющий свои собственные методы `readObject()` и `writeObject()`. Эти методы записывают дату в виде количества миллисекунд от начального момента отсчета времени (т.е. от полуночи по Гринвичу 1 января 1970 г.). Класс `Date` имеет сложное внутреннее представление, в котором хранится как объект типа `Calendar`, так и счетчик миллисекунд для оптимизации операций поиска. Состояние объекта типа `Calendar` избыточно и поэтому не требует обязательного сохранения. Методы `readObject()` и `writeObject()` должны сохранять и загружать поля данных только своего класса, не обращая особого внимания на данные из суперкласса или каких-нибудь других классов.

Вместо того чтобы сохранять и восстанавливать данные в объектах с помощью стандартного механизма сериализации, в классе можно определить свой механизм. Для этого класс должен реализовать интерфейс `Externalizable`. Это, в свою очередь, требует, чтобы в нем были также определены два следующих метода:

```
public void readExternal(ObjectInputStream in)
       throws IOException, ClassNotFoundException;
public void writeExternal(ObjectOutputStream out)
       throws IOException;
```

В отличие от методов `readObject()` и `writeObject()`, которые обсуждались в предыдущем разделе, эти методы сами полностью отвечают за сохранение и восстановление всего объекта *вместе* с данными суперкласса. Механизм сериализации просто фиксирует класс объекта в потоке ввода-вывода. При чтении объекта типа `Externalizable` поток ввода создает этот объект с помощью конструктора без аргументов и затем вызывает метод `readExternal()`. Ниже показано, как эти методы можно реализовать в классе `Employee`.

```
public void readExternal(ObjectInput s)
       throws IOException
{
  name = s.readUTF();
  salary = s.readDouble();
```

```
  hireDay = new Date(s.readLong());
}

public void writeExternal(ObjectOutput s)
       throws IOException
{
  s.writeUTF(name);
  s.writeDouble(salary);
  s.writeLong(hireDay.getTime());
}
```

ВНИМАНИЕ! В отличие от методов **readObject()** и **writeObject()**, которые являются закрытыми и могут вызываться только механизмом сериализации, методы **readExternal()** и **writeExternal()** являются открытыми. В частности, метод **readExternal()** допускает возможное изменение состояния существующего объекта.

2.3.4. Сериализация одноэлементных множеств и типизированных перечислений

Особого внимания требует сериализация и десериализация объектов, которые считаются единственными в своем роде. Обычно такое внимание требуется при реализации одноэлементных множеств (так называемых одиночек) и типизированных перечислений.

Так, если при написании программ на Java используется языковая конструкция `enum`, особенно беспокоиться по поводу сериализации не стоит, поскольку она будет произведена должным образом. Но что, если имеется некоторый унаследованный код, содержащий перечислимый тип вроде приведенного ниже.

```
public class Orientation
{
  public static final Orientation HORIZONTAL =
         new Orientation(1);
  public static final Orientation VERTICAL =
         new Orientation(2);

  private int value;

  private Orientation(int v) { value = v; }
}
```

Подобный подход очень широко применялся до того, как в Java были внедрены перечисления. Обратите внимание на закрытый характер конструктора. Это означает, что создать какие-нибудь другие объекты, помимо `Orientation.HORIZONTAL` и `Orientation.VERTICAL`, нельзя. А для выполнения проверки на равенство объектов можно использовать операцию ==, как показано ниже.

```
if (orientation == Orientation.HORIZONTAL) . . .
```

Но имеется одна важная особенность, о которой не следует забывать, когда типизированное перечисление реализует интерфейс `Serializable`. Применяемый по умолчанию механизм сериализации для этого не подходит. Попробуем, например, записать значение типа `Orientation` и затем прочитать его обратно следующим образом:

```
Orientation original = Orientation.HORIZONTAL;
ObjectOutputStream out = . . .;
out.write(original);
out.close();
ObjectInputStream in = . . .;
var saved = (Orientation) in.read();
```

Если затем произвести приведенную ниже проверку, она не пройдет. На самом деле переменная `saved` содержит совершенно новый объект типа `Orientation`, не равный ни одной из предопределенных констант. И несмотря на то что конструктор класса `Orientation` является закрытым, механизм сериализации все равно позволяет создавать новые объекты!

```
if (saved == Orientation.HORIZONTAL) . . .
```

В качестве выхода из этого затруднительного положения придется определить еще один специальный метод сериализации под названием `readResolve()`. В этом случае метод `readResolve()` вызывается после десериализации объекта. Он должен возвращать объект, превращаемый далее в значение, возвращаемое из метода `readObject()`. В рассматриваемом здесь примере метод `readResolve()` обследует поле `value` и возвратит соответствующую перечислимую константу, как показано ниже. Не следует, однако, забывать, что метод `readResolve()` нужно ввести во все типизированные перечисления в унаследованном коде, а также во все классы, где применяется проектный шаблон одиночки.

```
protected Object readResolve() throws ObjectStreamException
{
  if (value == 1) return Orientation.HORIZONTAL;
  if (value == 2) return Orientation.VERTICAL;
  return null; // этого не должно произойти!
}
```

2.3.5. Контроль версий

Если вы применяете механизм сериализации для сохранения объектов, вам придется заранее продумать все, что может произойти при последующем усовершенствовании вашей прикладной программы. В частности, сможет ли версия 1.1 вашей программы читать старые файлы и смогут ли пользователи, по-прежнему работающие с версией 1.0, читать файлы, которые производит новая версия? Конечно, было бы совсем неплохо, если бы файлы объектов могли успешно справляться с неизбежной эволюцией классов.

На первый взгляд, подобное кажется невозможным. Ведь если определение класса изменяется хоть каким-то образом, сразу же изменяется и его "отпечаток" SHA, а, как вам должно быть уже известно, потоки ввода объектов откажутся читать объекты с отличающимися "отпечатками". Но в то же время класс может уведомить, что он совместим со своей более ранней версией. А для этого следует прежде всего получить "отпечаток" более ранней версии класса. Это можно сделать с помощью входящей в состав JDK автономной утилиты `serialver`. Например, выполнение команды

```
serialver Employee
```

даст такой результат:

```
Employee: static final long serialVersionUID =
        -1814239825517340645L;
```

Во всех *последующих* версиях класса константа `serialVersionUID` должна определяться точно с таким же "отпечатком", как и в исходной версии:

```
class Employee implements Serializable // версия 1.1
{
  . . .
  public static final long serialVersionUID =
         -1814239825517340645L;
}
```

При наличии в классе статического члена данных `serialVersionUID` он не станет вычислять "отпечаток" вручную, а просто воспользуется значением, содержащимся в этом члене. А после размещения такого статического члена в классе система сериализации будет сразу же готова к чтению разных версий объектов данного класса.

Если изменяются только методы класса, то никаких осложнений при чтении данных о новых объектах не возникнет. А если изменения произойдут в полях данных, некоторые осложнения все же могут возникнуть. Например, старый файловый объект может содержать больше или меньше полей данных, чем тот, который применяется в программе в данный момент, да и типы полей данных могут отличаться. В таком случае поток ввода объектов попытается привести потоковый объект к текущей версии класса.

Поток ввода объектов сравнит поля данных из текущей версии класса с полями данных из той версии класса, которая указана в потоке. Конечно, принимать во внимание он будет только непереходные и нестатические поля данных. При совпадении имен, но несовпадении типов полей поток ввода объектов, естественно, не будет пытаться преобразовывать один тип данных в другой, а следовательно, такие объекты будут считаться несовместимыми. При наличии у объекта в потоке ввода таких полей данных, которых нет в текущей версии, поток ввода объектов будет игнорировать их. А при наличии в текущей версии таких полей, которых нет в потоковом объекте, для всех дополнительных полей будет устанавливаться соответствующее им значение по умолчанию (для объектов это пустое значение `null`, для чисел — `0`, для логических значений — `false`).

Обратимся к конкретному примеру. Допустим, ряд записей о сотрудниках был сохранен на диске с помощью исходной версии класса `Employee` (версии 1.0), а затем она была заменена версией 2.0 данного класса, в которой введено дополнительное поле данных `department`. На рис. 2.7 показано, что при чтении объекта версии 1.0 в программе, где используются объекты версии 2.0, произойдет следующее: в поле `department` будет установлено пустое значение `null`. А на рис. 2.8 показана обратная ситуация, возникающая при чтении объекта версии 2.0 в программе, где используются объекты версии 1.0. В этом случае дополнительное поле `department` будет проигнорировано.

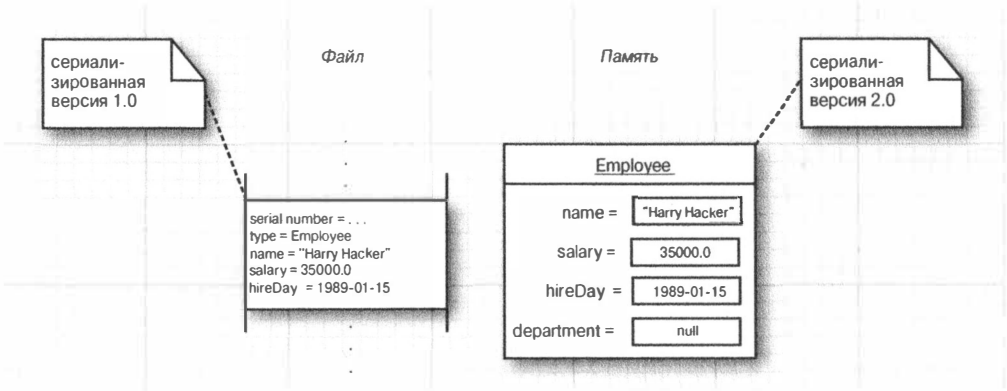


Рис. 2.7. Чтение объекта с меньшим количеством полей данных

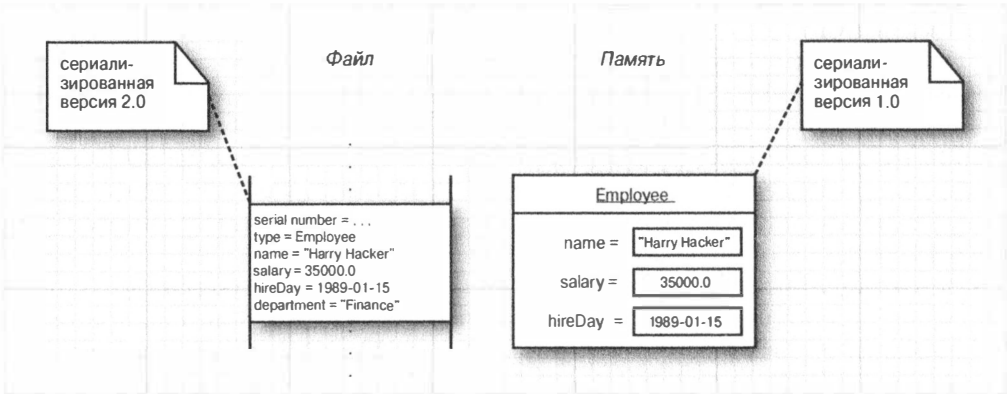


Рис. 2.8. Чтение объекта с большим количеством полей данных

Насколько безопасным окажется данный процесс? Все зависит от обстоятельств. Игнорирование поля данных кажется безвредным. Ведь у получателя все равно остаются те же данные, с которыми он знает, как обращаться. В то же время установка в поле данных пустого значения `null` может и не быть безопасной. Во многих классах делается немало для инициализации всех полей данных во всех конструкторах значениями, отличающимися от `null`, чтобы не предусматривать заранее в методах обработку пустых данных. Следовательно, разработчик классов должен сам решить, что лучше: реализовать дополнительный код в методе `readObject()` для устранения препятствий на пути к совместимости версий или же обеспечить достаточную надежность методов для успешной обработки пустых данных. Разработчик класса должен реализовать дополнительный код в методе `readObject()` для устранения несовместимости версий или обеспечения достаточной надежности методов при обработке данных типа `null`.

СОВЕТ. Прежде чем вводить поле **`serialVersionUID`** в класс, необходимо выяснить, зачем вообще этот класс сделан сериализируемым. Так, если сериализация служит только для кратковременного хранения данных (например, распределенных вызовов методов на сервере приложений), то никакого контроля версий и поля **`serialVersionUID`** вообще не требуется.

Как, впрочем, и в том случае, если сериализируемый класс расширяется, но сохранение его экземпляров не предполагается. Если же интегрированная среда разработки выдает досадные предупреждения относительно сериализации, их можно подавить или же ввести в исходный код аннотацию **@SuppressWarnings("serial")**. Это намного надежнее, чем вводить поле **serialVersionUID**, об изменении содержимого которого можно впоследствии просто забыть.

2.3.6. Применение сериализации для клонирования

Механизм сериализации находит еще одно интересное применение: он позволяет легко клонировать объект, при условии, что класс последнего является сериализируемым. Для этого достаточно сериализовать объект в поток вывода, а затем прочитать его обратно. В результате получится новый объект, представляющий собой точную (полную) копию уже существующего. Записывать этот объект в файл совсем не обязательно. Вместо этого можно воспользоваться потоком вывода типа `ByteArrayOutputStream`, сохранив данные в байтовом массиве.

Как демонстрируется в примере программы из листинга 2.4, чтобы получить клон объекта, достаточно расширить класс `SerialCloneable`. Следует, однако, иметь в виду, что, несмотря на всю изощренность такого способа, он, как правило, оказывается менее быстродействующим, чем способ клонирования, явным образом создающий новый объект и копирующий или клонирующий поля данных.

Листинг 2.4. Исходный код из файла **serialClone/SerialCloneTest.java**

```
1  package serialClone;
2
3  /**
4   * @version 1.22 2018-05-01
5   * @author Cay Horstmann
6   */
7
8  import java.io.*;
9  import java.time.*;
10
11 public class SerialCloneTest
12 {
13   public static void main(String[] args)
14         throws CloneNotSupportedException
15   {
16     var harry = new Employee("Harry Hacker",
17                              35000, 1989, 10, 1);
18     // клонировать объект harry
19     var harry2 = (Employee) harry.clone();
20
21     // модифицировать объект harry
22     harry.raiseSalary(10);
23
24     // теперь оригинал и клон объекта harry отличаются
25     System.out.println(harry);
26     System.out.println(harry2);
27   }
28 }
```

```
29
30  /**
31   * Класс, в методе клонирования которого
32   * применяется сериализация
33   */
34  class SerialCloneable implements Cloneable, Serializable
35  {
36    public Object clone()
37          throws CloneNotSupportedException
38    {
39      try {
40        // сохранить объект в массиве байтов
41        var bout = new ByteArrayOutputStream();
42        try (var out = new ObjectOutputStream(bout))
43        {
44          out.writeObject(this);
45        }
46
47        // ввести клон объекта из массива байтов
48        try (var bin = new ByteArrayInputStream(
49                           bout.toByteArray()))
50        {
51          var in = new ObjectInputStream(bin);
52          return in.readObject();
53        }
54      }
55      catch (IOException | ClassNotFoundException e)
56      {
57        var e2 = new CloneNotSupportedException();
58        e2.initCause(e);
59        throw e2;
60      }
61    }
62  }
63
64  /**
65   * Класс Employee class, переопределяемый для
66   * расширения класса SerialCloneable
67   */
68  class Employee extends SerialCloneable
69  {
70    private String name;
71    private double salary;
72    private LocalDate hireDay;
73
74    public Employee(String n, double s, int year,
75                    int month, int day)
76    {
77      name = n;
78      salary = s;
79      hireDay = LocalDate.of(year, month, day);
80    }
81
82    public String getName()
83    {
84      return name;
```

```
85    }
86
87    public double getSalary()
88    {
89      return salary;
90    }
91
92    public LocalDate getHireDay()
93    {
94      return hireDay;
95    }
96
97    /**
98     * Поднимает зарплату данному работнику
99     * @byPercent Процент повышения зарплаты
100    */
101   public void raiseSalary(double byPercent)
102   {
103     double raise = salary * byPercent / 100;
104     salary += raise;
105   }
106
107   public String toString()
108   {
109     return getClass().getName()
110           + "[name=" + name
111           + ",salary=" + salary
112           + ",hireDay=" + hireDay
113           + "]";
114   }
115 }
```

2.4. Манипулирование файлами

Ранее в этой главе уже пояснялось, как читать и записывать данные в файл. Но управление файлами не ограничивается только чтением и записью. Классы `Path` и `Files` инкапсулируют все функциональные возможности, которые могут потребоваться для работы с файловой системой на машине пользователя. Так, с помощью класса `Files` можно выяснить время последнего изменения, удалить или переименовать файлы. Иными словами, классы потоков ввода-вывода служат для манипулирования содержимым файлов, а классы, рассматриваемые в этом разделе, — для хранения файлов на диске.

Классы `Path` и `Files` были внедрены в версии Java 7. Они намного удобнее класса `File`, внедренного еще в версии JDK 1.0. Есть все основания полагать, что они найдут широкое признание у программирующих на Java. Именно поэтому они и рассматриваются в этом разделе.

2.4.1. Пути к файлам

Путь представляет собой последовательность имен каталогов, после которой следует (хотя и не обязательно) имя файла. Первой составляющей пути может

быть корневой каталог, например / или C:\. В зависимости от конкретной операционной системы в пути допускаются разные составляющие. Если путь начинается с корневого каталога, он считается *абсолютным*, а иначе — *относительным*. В качестве примера в приведенном ниже фрагменте кода составляется абсолютный и относительный путь. При составлении абсолютного пути предполагается наличие UNIX-подобной файловой системы.

```
Path absolute = Paths.get("/home", "cay");
Path relative = Paths.get("myprog", "conf",
                          "user.properties");
```

Статический метод `Paths.get()` получает в качестве своих параметров одну или несколько символьных строк, соединяя их через разделитель пути, используемый по умолчанию в данной файловой системе (знак / — в UNIX-подобной файловой системе или знак \ — в Windows). Затем в этом методе осуществляется синтаксический анализ результата, и если путь оказывается недостоверным для данной файловой системы, то генерируется исключение типа `InvalidPathException`, в противном случае получается объект типа `Path`.

В качестве параметра методу `get()` можно передать единственную символьную строку, содержащую несколько составляющих пути. Например, путь можно прочитать из конфигурационного файла следующим образом:

```
String baseDir = props.getProperty("base.dir")
    // Содержимое переменной baseDir может быть
    // представлено следующей символьной строкой:
    // "opt/myprog" или "c:\Program Files\myprog"
Path basePath = Paths.get(baseDir); // в переменной baseDir
                                    // допускаются разделители пути
```

НА ЗАМЕТКУ! Путь совсем не обязательно должен приводить к уже существующему файлу. Ведь он представляет собой не более чем абстрактную последовательность имен. Как поясняется в следующем разделе, при создании файла сначала составляется путь к нему, а затем вызывается метод для создания соответствующего файла.

Зачастую пути объединяются, или *разрешаются*. Так, в результате вызова `p.resolve(q)` возвращается путь по следующим правилам.

- Если `q` — абсолютный путь, то результат равен `q`.
- В противном случае результат равен "`p` затем `q`" по правилам, принятым в данной файловой системе.

Допустим, в прикладной программе требуется найти рабочий каталог относительно заданного базового каталога, читаемого из конфигурационного файла, как было показано в предыдущем примере кода. Для этой цели служит приведенный ниже фрагмент кода.

```
Path workRelative = Paths.get("work");
Path workPath = basePath.resolve(workRelative);
```

Имеется сокращенный вариант метода `resolve()`, принимающий в качестве своего параметра символьную строку вместо пути, как показано ниже.

```
Path workPath = basePath.resolve("work");
```

Кроме того, имеется удобный метод `resolveSibling()`, разрешающий путь относительно его родителя, порождая родственный путь. Так, если `/opt/myapp/work` — путь к рабочему каталогу, то в результате следующего вызова образуется путь `/opt/myapp/temp`:

```
Path tempPath = workPath.resolveSibling("temp")
```

Противоположную методу `resolve()` функцию выполняет метод `relativize()`. В результате вызова `p.relativize(r)` порождается путь `q`, который, в свою очередь, порождает путь `r` при своем разрешении по исходному пути `p`. Например, в результате релятивизации пути `"/home/harry"` относительно пути `"/home/fred/input.txt"` порождается относительный путь `"../fred/input.txt"`. В данном случае две точки (`..`) обозначают родительский каталог в файловой системе.

Метод `normalize()` удаляет любые избыточные знаки `.` и `..` или иные составляющие пути, которые считаются лишними в файловой системе. Например, в результате нормализации пути `/home/harry /../fred/./input.txt` получается путь `/home/fred/input.txt`. А метод `toAbsolutePath()` порождает абсолютный путь из заданного пути, начиная с коренной составляющей, например `/home/fred/input.txt` или `c:\Users\fred\input.txt`.

В интерфейсе `Path` имеется немало полезных методов для разделения путей к файлам на составляющие. В приведенном ниже фрагменте кода демонстрируется применение наиболее полезных методов из этого интерфейса.

```
Path p = Paths.get("/home", "fred", "myprog.properties");
Path parent = p.getParent(); // путь /home/fred
Path file = p.getFileName(); // путь myprog.properties
Path root = p.getRoot();     // путь /
```

Как пояснялось в первом томе настоящего издания, из объекта типа `Path` можно построить объект типа `Scanner` следующим образом:

```
var in = new Scanner(Paths.get("/home/fred/input.txt"));
```

НА ЗАМЕТКУ! Время от времени вам, возможно, придется иметь дело с унаследованными прикладными интерфейсами API, где вместо интерфейса **`Path`** применяется класс **`File`**. Так, в интерфейсе **`Path`** имеется метод **`toFile()`**, тогда как в классе **`File`** — метод **`toPath()`**.

`java.nio.file.Paths` 7

- **`static Path get(String first, String... more)`**
  Составляет путь, соединяя заданные символьные строки.

`java.nio.file.Path` 7

- **`Path resolve(Path other)`**
- **`Path resolve(String other)`**
  Если параметр ***`other`*** содержит абсолютный путь, то возвращается путь ***`other`***, а иначе — путь, получаемый в результате объединения путей **`this`** и ***`other`***.

java.nio.file.Path 7 *(окончание)*

- **Path resolveSibling(Path *other*)**
- **Path resolveSibling(String *other*)**

  Если параметр ***other*** содержит абсолютный путь, то возвращается путь ***other***, а иначе — путь, получаемый в результате объединения родителя пути **this** и заданного пути ***other***.
- **Path relativize(Path *other*)**

  Возвращает относительный путь, порождающий путь ***other*** при разрешении пути **this**.
- **Path normalize()**

  Удаляет из пути избыточные составляющие, например знаки `.` и `...`
- **Path toAbsolutePath()**

  Возвращает абсолютный путь, равнозначный данному пути.
- **Path getParent()**

  Возвращает родительский путь или пустое значение **null**, если у данного пути отсутствует родительский путь.
- **Path getFileName()**

  Возвращает последнюю составляющую данного пути или пустое значение **null**, если у данного пути отсутствуют составляющие.
- **Path getRoot()**

  Возвращает корневую составляющую данного пути или пустое значение **null**, если у данного пути отсутствует корневая составляющая.
- **toFile()**

  Составляет объект типа **File** из данного пути.

java.io.File 1.0

- **Path toPath()** 7

  Составляет объект типа **Path** из данного пути.

2.4.2. Чтение и запись данных в файлы

Класс `Files` упрощает и ускоряет выполнение типичных операций над файлами. Например, содержимое всего файла нетрудно прочитать следующим образом:

```
byte[] bytes = Files.readAllBytes(path);
```

Если же требуется прочитать содержимое файла в виде символьной строки, сначала необходимо вызвать метод `readAllBytes()`, как показано выше, а затем следующий конструктор:

```
var content = new String(bytes, charset);
```

Но если требуется получить содержимое файла в виде последовательности строк, то достаточно сделать следующий вызов:

```
List<String> lines = Files.readAllLines(path, charset);
```

С другой стороны, если требуется записать в файл символьную строку, достаточно сделать вызов

```
Files.write(path, content.getBytes(charset));
```

А для присоединения строки к файлу можно сделать такой вызов:

```
Files.write(path, content.getBytes(charset),
            StandardOpenOption.APPEND);
```

Кроме того, в файл можно записать целый ряд строк следующим образом:

```
Files.write(path, lines);
```

Приведенные выше простые методы предназначены для манипулирования текстовыми файлами умеренной длины. Если же файл крупный или двоичный, то для манипулирования им можно воспользоваться упоминавшимися ранее потоками ввода-вывода или чтения и записи данных, как показано ниже. Имеющиеся у них удобные методы избавляют от необходимости обращаться непосредственно к классам `FileInputStream`, `FileOutputStream`, `BufferedReader` или `BufferedWriter`.

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
Reader in = Files.newBufferedReader(path, charset);
Writer out = Files.newBufferedWriter(path, charset);
```

java.nio.file.Files 7

- **static byte[] readAllBytes(Path *path*)**
- **static List<String> readAllLines(Path *path*, Charset *charset*)**

  Читают содержимое файла.
- **static Path write(Path *path*, byte[] *contents*, OpenOption... *options*)**
- **static Path write(Path *path*, Iterable<? extends CharSequence> *contents*, OpenOption *options*)**

  Записывают заданное содержимое в файл и возвращают ***path*** как путь к нему.
- **static InputStream newInputStream(Path *path*, OpenOption... *options*)**
- **static OutputStream newOutputStream(Path *path*, OpenOption... *options*)**
- **static BufferedReader newBufferedReader(Path *path*, Charset *charset*)**
- **static BufferedWriter newBufferedWriter(Path *path*, Charset *charset*, OpenOption... *options*)**

  Открывают файл для чтения или записи.

2.4.3. Создание файлов и каталогов

Чтобы создать новый каталог, достаточно сделать следующий вызов:

```
Files.createDirectory(path);
```

Все составляющие пути к каталогу, кроме последней, должны уже существовать. А для создания промежуточных каталогов достаточно сделать такой вызов:

```
Files.createDirectories(path);
```

Чтобы создать пустой файл, следует сделать приведенный ниже вызов.

```
Files.createFile(path);
```

Если файл уже существует, то в результате данного вызова генерируется исключение. Поэтому, выполняя операцию создания файла, следует проверять ее атомарность и факт существования файла. Если файл не существует, он создается, прежде чем у кого-нибудь другого появится возможность сделать то же самое.

Для создания временного файла или каталога в указанном месте файловой системы имеются удобные методы. Примеры их применения демонстрируются в приведенном ниже фрагменте кода, где `dir` — это объект типа `Path`, а `prefix`/`suffix` — символьные строки, которые могут быть нулевыми (`null`). Например, в результате вызова `Files.createTempFile(null, ".txt")` может быть возвращен путь `/tmp/12344055223648371941.txt`.

```
Path newPath = Files.createTempFile(dir, prefix, suffix);
Path newPath = Files.createTempFile(prefix, suffix);
Path newPath = Files.createTempDirectory(dir, prefix);
Path newPath = Files.createTempDirectory(prefix);
```

При создании файла или каталога можно также указать его атрибуты, в том числе владельцев или права доступа. Но конкретные подробности зависят от применяемой файловой системы, и поэтому здесь они не рассматриваются.

java.nio.file.Files 7

- **static Path createFile(Path *path*, FileAttribute<?>... *attrs*)**
- **static Path createDirectory(Path *path*, FileAttribute<?>... *attrs*)**
- **static Path createDirectories(Path *path*, FileAttribute<?>... *attrs*)**

  Создают файл или каталог. В частности, метод **createDirectories()** создает также любые промежуточные каталоги.
- **static Path createTempFile(String *prefix*, String *suffix*, FileAttribute<?>... *attrs*)**
- **static Path createTempFile(Path *parentDir*, String *prefix*, String *suffix*, FileAttribute<?>... *attrs*)**
- **static Path createTempDirectory(String *prefix*, FileAttribute<?>... *attrs*)**
- **static Path createTempDirectory(Path *parentDir*, String *prefix*, FileAttribute<?>... *attrs*)**

  Создают временный файл или каталог в месте, пригодном для хранения временных файлов, или же в заданном родительском каталоге. Возвращают путь к созданному файлу или каталогу.

2.4.4. Копирование, перемещение и удаление файлов

Чтобы скопировать файл из одного места в другое, достаточно сделать следующий вызов:

```
Files.copy(fromPath, toPath);
```

А для того чтобы переместить файл, т.е. сделать его копию и удалить оригинал, следует сделать вызов

```
Files.move(fromPath, toPath);
```

Исход операции копирования или перемещения файлов окажется неудачным, если целевой файл существует. Если же требуется перезаписать целевой файл, при вызове соответствующего метода следует указать дополнительный параметр REPLACE_EXISTING, а если требуется скопировать все атрибуты файла — дополнительный параметр COPY_ATTRIBUTES. Кроме того, можно указать оба дополнительных параметра следующим образом:

```
Files.copy(fromPath, toPath,
           StandardCopyOption.REPLACE_EXISTING,
           StandardCopyOption.COPY_ATTRIBUTES);
```

Имеется также возможность сделать операцию перемещения атомарной. Этим гарантируется, что операция перемещения завершится успешно или что источник данных продолжает существовать. В приведенной ниже строке кода показано, каким образом для этой цели используется дополнительный параметр ATOMIC_MOVE.

```
Files.move(fromPath, toPath,
           StandardCopyOption.ATOMIC_MOVE);
```

В файл, находящийся по пути, указанному в объекте типа Path, можно также скопировать поток ввода. Это, по существу, означает сохранение потока ввода на диске. Аналогично содержимое файла, находящегося по пути, указанному в объекте типа Path, можно направить в поток вывода. В приведенном ниже фрагменте кода показано, каким образом выполняются обе эти операции. Как и в других вызовах метода copy(), можно предоставить по мере надобности дополнительные параметры.

```
Files.copy(inputStream, toPath);
Files.copy(fromPath, outputStream);
```

Наконец, для удаления файла достаточно вызвать следующий метод:

```
Files.delete(path);
```

Этот метод генерирует исключение, если файл не существует. Поэтому вместо него, возможно, придется вызвать приведенный ниже метод. Оба метода можно также использовать для удаления пустого каталога.

```
boolean deleted = Files.deleteIfExists(path);
```

В табл. 2.3 сведены дополнительные параметры, доступные при выполнении операций с файлами.

Таблица 2.3. Стандартные параметры для операций с файлами

Параметр	Описание
StandardOpenOption; применяется в потоках ввода-вывода типа **newBufferedWriter**, **newInputStream**, **newOutputStream** для операции записи	
READ	Открыть файл для чтения
WRITE	Открыть файл для записи

Окончание табл. 2.3

Параметр	Описание
APPEND	Если файл открыт для записи, присоединить данные в конце этого файла
TRUNCATE_EXISTING	Если файл открыт для записи, удалить его текущее содержимое
CREATE_NEW	Создать новый файл, указать на неудачный исход операции, если файл уже существует
CREATE	Создать файл атомарно, если файл не существует
DELETE_ON_CLOSE	Удалить файл наилучшим образом после его закрытия
SPARSE	Указать файловой системе, что этот файл окажется разреженным
DSYNC|SYNC	Потребовать, чтобы при каждом обновлении файла данные и метаданные синхронно записывались на запоминающем устройстве
StandardCopyOption; применяется в операциях копирования и перемещения	
ATOMIC_MOVE	Переместить файл атомарно
COPY_ATTRIBUTES	Скопировать атрибуты файла
REPLACE_EXISTING	Заменить целевой файл, если он существует
LinkOption; применяется во всех упомянутых выше методах, а также в методах **exists()**, **isDirectory()**, **isRegularFile()**	
NOFOLLOW_LINKS	Не следовать по символическим ссылкам
FileVisitOption; применяется в методах **find()**, **walk()**, **walkFileTree()**	
FOLLOW_LINKS	Следовать по символическим ссылкам

java.nio.file.Files 7

- **static Path copy(Path *from*, Path *to*, CopyOption... *options*)**
- **staticPath move(Path *from*, Path *to*, CopyOption... *options*)**

  Копируют или перемещают файл из исходного места ***from*** в заданное целевое место ***to***, возвращая последнее.
- **static long copy(InputStream *from*, Path *to*, CopyOption... *options*)**
- **static long copy(Path *from*, OutputStream *to*, CopyOption... *options*)**

  Копируют данные в файл из потока ввода или из файла в поток вывода, возвращая количество скопированных байтов.
- **static void delete(Path *path*)**
- **static boolean deleteIfExists(Path *path*)**

  Удаляют заданный файл или пустой каталог. Первый метод генерирует исключение, если заданный файл или каталог не существует. Второй метод возвращает в этом случае логическое значение **false**.

2.4.5. Получение сведений о файлах

Ниже перечислены методы, возвращающие логическое значение для проверки свойства пути.

- `exists()`
- `isHidden()`

- `isReadable()`, `isWritable()`, `isExecutable()`
- `isRegularFile()`, `isDirectory()`, `isSymbolicLink()`

Метод `size()` возвращает количество байтов в файле, как показано в приведенной ниже строке кода. А метод `getOwner()` возвращает владельца файла в виде экземпляра класса `java.nio.file.attribute.UserPrincipal`.

```
long fileSize = Files.size(path);
```

Все файловые системы уведомляют об основных атрибутах, инкапсулированных в интерфейсе `BasicFileAttributes`. Они частично перекрывают прочие сведения о файлах. Ниже перечислены основные атрибуты файлов.

- Моменты времени, когда файл был создан, последний раз открывался и видоизменялся, в виде экземпляров класса `java.nio.file.attribute.FileTime`.
- Является ли файл обычным, каталогом, символической ссылкой или ничем из перечисленного.
- Размер файла.
- Файловый ключ — объект некоторого класса, характерный для применяемой файловой системы и способный (или не способный) однозначно определять файл.

Для получения перечисленных выше атрибутов файлов достаточно сделать следующий вызов:

```
BasicFileAttributes attributes = files.readAttributes(
                    path, BasicFileAttributes.class);
```

Если заранее известно, что пользовательская файловая система соответствует стандарту POSIX, в таком случае можно получить экземпляр класса `PosixFileAttributes` следующим образом:

```
PosixFileAttributes attributes = files.readAttributes(
                    path, PosixFileAttributes.class);
```

Далее можно определить группового или индивидуального владельца файла, а также права на групповой или глобальный доступ к нему. Мы не будем здесь вдаваться в подробности этого процесса, поскольку большая часть сведений о файлах не переносится из одной операционной системы в другую.

java.nio.file.Files 7

- **static boolean exists(Path *path*)**
- **static boolean isHidden(Path *path*)**
- **static boolean isReadable(Path *path*)**
- **static boolean isWritable(Path *path*)**
- **static boolean isExecutable(Path *path*)**
- **static boolean isRegularFile(Path *path*)**

java.nio.file.Files 7 *(окончание)*

- **static boolean isDirectory(Path *path*)**
- **static boolean isSymbolicLink(Path *path*)**
  Проверяют заданное свойство файла по указанному пути.
- **static long size(Path *path*)**
  Получает размер файла в байтах.
- **A readAttributes(Path *path*, Class<A> *type*, LinkOption... *options*)**
  Читает атрибуты файла, относящиеся к типу **A**.

java.nio.file.attribute.BasicFileAttributes 7

- **FileTime creationTime()**
- **FileTime lastAccessTime()**
- **FileTime lastModifiedTime()**
- **boolean isRegularFile()**
- **boolean isDirectory()**
- **boolean isSymbolicLink()**
- **long size()**
- **Object fileKey()**
  Получают запрашиваемый атрибут файла.

2.4.6. Обход элементов каталога

Статический метод `Files.list()` возвращает поток данных типа `Stream<Path>`, откуда читаются элементы каталога. Содержимое каталога читается по требованию, и благодаря этому становится возможной эффективная обработка каталогов с большим количеством элементов.

Для чтения содержимого каталога требуется закрыть системные ресурсы, поэтому данную операцию необходимо заключить в блок оператора `try` следующим образом:

```
try (Stream<Path> entries = Files.list(pathToDirectory))
{
  . . .
}
```

Метод `list()` не входит в подкаталоги. Чтобы обработать все порожденные элементы каталога, следует воспользоваться методом `Files.walk()`:

```
try (Stream<Path> entries = Files.walk(pathToRoot))
{
  // Содержит все порожденные элементы,
  // обойденные в глубину
}
```

Ниже приведен пример обхода дерева каталогов из разархивированного файла `src.zip`. Как видите, всякий раз, когда в результате обхода получается каталог, происходит вход в него, прежде чем продолжать обход родственных ему каталогов.

```
java
java/nio
java/nio/DirectCharBufferU.java
java/nio/ByteBufferAsShortBufferRL.java
java/nio/MappedByteBuffer.java
. . .
java/nio/ByteBufferAsDoubleBufferB.java
java/nio/charset
java/nio/charset/CoderMalfunctionError.java
java/nio/charset/CharsetDecoder.java
java/nio/charset/UnsupportedCharsetException.java
java/nio/charset/spi
java/nio/charset/spi/CharsetProvider.java
java/nio/charset/StandardCharsets.java
java/nio/charset/Charset.java
. . .
java/nio/charset/CoderResult.java
java/nio/HeapFloatBufferR.java
. . .
```

Глубину дерева каталогов, которое требуется обойти, можно ограничить, вызвав метод `Files.walk(pathToRoot, depth)`. В обоих рассматриваемых здесь вызовах метода `walk()` имеются аргументы переменной длины типа `FileVisitOption...`, но для перехода по символическим ссылкам можно предоставить только параметр `FOLLOW_LINKS`.

НА ЗАМЕТКУ! Чтобы отфильтровать пути, возвращаемые методом **`walk()`** по критерию, включающему в себя атрибуты файлов, хранящиеся в каталоге, в том числе размер, время создания или тип (файла, каталога, символической ссылки), вместо метода **`walk()`** лучше воспользоваться методом **`find()`**. Этот метод следует вызывать с предикатной функцией, принимающей в качестве параметров путь и объект типа **`BasicFileAttributes`**. Единственное преимущество такого подхода состоит в его эффективности. Ведь чтение каталога происходит в любом случае, и поэтому атрибуты сразу же становятся доступными.

В следующем фрагменте кода метод `Files.walk()` применяется для копирования одного каталога в другой:

```
Files.walk(source).forEach(p ->
  {
    try {
      Path q = target.resolve(source.relativize(p));
      if (Files.isDirectory(p))
        Files.createDirectory(q);
      else
        Files.copy(p, q);
    } catch (IOException ex) {
      throw new UncheckedIOException(ex);
    }
  });
```

К сожалению, методом `Files.walk()` не так-то просто воспользоваться для удаления дерева каталогов, поскольку для этого нужно обойти все порожденные элементы, прежде чем удалить родительский элемент. В следующем разделе поясняется, как преодолеть подобное затруднение.

2.4.7. Применение потоков каталогов

Как пояснялось в предыдущем разделе, метод `Files.walk()` возвращает поток данных типа `Stream<Path>` для обхода порожденных элементов каталога. Но иногда требуется более точный контроль над процессом обхода элементов каталога. И в таком случае следует воспользоваться потоком данных типа `Files.newDirectoryStream`, который производит поток данных типа `DirectoryStream`. Однако он не относится к интерфейсу, подчиненному интерфейсу `java.util.stream.Stream`, предназначенному для обхода элементов каталога. Напротив, он относится к интерфейсу, подчиненному интерфейсу `Iterable`, что дает возможность использовать поток каталогов в расширенном цикле `for`, как демонстрируется в следующем примере кода:

```
try (DirectoryStream<Path> entries =
     Files.newDirectoryStream(dir))
{
  for (Path entry : entries)
   обработать entries
}
```

Блок оператора `try` с ресурсами обеспечивает надлежащее закрытие потока ввода из каталога. Определенного порядка обхода элементов каталога не существует. Файлы можно отфильтровать по глобальному шаблону, как показано ниже. Все глобальные шаблоны перечислены в табл. 2.4.

```
try (DirectoryStream<Path> entries =
     Files.newDirectoryStream(dir, "*.java"))
```

Таблица 2.4. Глобальные шаблоны

Шаблон	Описание	Пример применения
`*`	Совпадает со всеми нулями и дополнительными символами в составляющей пути	Шаблон **`*.java`** совпадает со всеми файлами исходного кода на Java в текущем каталоге
`**`	Совпадает со всеми нулями и дополнительными символами, пересекая границы каталогов	Шаблон **`**.java`** совпадает со всеми файлами исходного кода на Java в любом подкаталоге
`?`	Совпадает с одним символом	Шаблон **`????.java`** совпадает со всеми файлами исходного кода на Java, имена которых состоят из четырех символов, не считая расширения
`[...]`	Совпадает с рядом символов, указываемых через дефис или со знаком отрицания: **`[0-9]`** или **`[!0-9]`** соответственно	Шаблон **`Test[0-9A-F].java`** совпадает с файлом **`Testx.java`**, где ***x*** — одна шестнадцатеричная цифра
`{...}`	Совпадает с альтернативными символами, разделенными запятыми	Шаблон **`*.{java,class}`** совпадает со всеми файлами исходного кода и классов на Java
`\`	Экранирует любые перечисленные выше шаблоны	Шаблон **`*\**`** совпадает со всеми файлами, в именах которых содержится знак `*`

ВНИМАНИЕ! Используя синтаксис глобальных шаблонов в Windows, экранируйте символы обратной косой черты дважды: один раз — в синтаксисе глобального шаблона, а другой раз — в синтаксисе символьной строки: **`Files.newDirectoryStream(dir, "C:\\\\")`**.

Если требуется обойти порожденные элементы каталога, следует вызвать метод `walkFileTree()` с объектом типа `FileVisitor` в качестве параметра. Этот объект уведомляется в следующих случаях.

- Когда встречается файл или каталог: `FileVisitResult visitFile(T path, BasicFileAttributes attrs)`.
- До обработки каталога: `FileVisitResult preVisitDirectory(T dir, IOException ex)`.
- После обработки каталога: `FileVisitResult postVisitDirectory(T dir, IOException ex)`.
- Когда возникает ошибка в связи с попыткой обратиться к файлу или каталогу, например, открыть каталог без надлежащих прав доступа: `FileVisitResult visitFileFailed(T path, IOException ex)`.

В каждом случае можно указать следующее.

- Продолжить обращение к следующему файлу: `FileVisitResult.CONTINUE`.
- Продолжить обход, не обращаясь к элементам каталога: `FileVisitResult.SKIP_SUBTREE`.
- Продолжить обход, не обращаясь к элементам каталога, родственным данному файлу: `FileVisitResult.SKIP_SIBLINGS`.
- Завершить обход: `FileVisitResult.TERMINATE`.

Если любой из методов генерирует исключение, обход каталогов завершается и данное исключение генерируется далее в методе `walkFileTree()`.

НА ЗАМЕТКУ! Несмотря на то что интерфейс **`FileVisitor`** относится к обобщенному типу, вам вряд ли придется пользоваться какой-нибудь другой его разновидностью, кроме **`FileVisitor<Path>`**. Метод **`walkFileTree()`** охотно принимает в качестве своего параметра объект обобщенного типа **`FileVisitor<? super Path>`**, но у типа **`Path`** не так уж и много супертипов.

Удобный класс `SimpleFileVisitor` реализует интерфейс `FileVisitor`. Все его методы, кроме `visitFileFailed()`, ничего особенного не делают, лишь продолжая выполнение. А метод `visitFileFailed()` генерирует исключение, возникающее в результате сбоя и, следовательно, прекращающее обход каталога. В качестве примера в приведенном ниже фрагменте кода демонстрируется, каким образом выводятся все подкаталоги из заданного каталога.

```
Files.walkFileTree(Paths.get("/"),
                   new SimpleFileVisitor<Path>()
{
  public FileVisitResult preVisitDirectory(Path path,
         BasicFileAttributes attrs) throws IOException
  {
    System.out.println(path);
    return FileVisitResult.CONTINUE;
  }
  public FileVisitResult postVisitDirectory(Path dir,
```

```
        IOException exc)
  {
    return FileVisitResult.CONTINUE;
  }
  public FileVisitResult visitFileFailed(Path path,
        IOException exc) throws IOException
  {
    return FileVisitResult.SKIP_SUBTREE;
  }
});
```

Следует, однако, иметь в виду, что методы `postVisitDirectory()` и `visitFileFailed()` придется переопределить, иначе обращение к файлам сразу же завершится аварийно, как только встретится каталог, который не разрешается открывать. И кроме того, атрибуты пути передаются методам `postVisitDirectory()` и `visitFile()` в качестве параметра. При обходе каталога уже пришлось обратиться к операционной системе для получения атрибутов, поскольку нужно каким-то образом отличать файлы от каталогов. Благодаря этому исключается необходимость делать еще один вызов.

Остальные методы из интерфейса `FileVisitor` оказываются полезными в том случае, если требуется выполнить служебные операции для входа и выхода из каталога. Например, при удалении дерева каталогов после всех файлов необходимо удалить и каталог, в котором они находились. Ниже приведен полный пример кода для удаления дерева каталогов.

```
// удалить дерево каталогов, начиная с корневого каталога
Files.walkFileTree(root, new SimpleFileVisitor<Path>()
{
  public FileVisitResult visitFile(Path file,
        BasicFileAttributes attrs) throws IOException
  {
    Files.delete(file);
    return FileVisitResult.CONTINUE;
  }
  public FileVisitResult postVisitDirectory(Path dir,
                        IOException e) throws IOException
  {
    if (e != null) throw e;
    Files.delete(dir);
    return FileVisitResult.CONTINUE;
  }
});
```

java.nio.file.Files 7

- **static DirectoryStream<Path> newDirectoryStream(Path *path*)**
- **static DirectoryStream<Path> newDirectoryStream(Path *path*, String *glob*)**

  Получают итератор для обхода всех файлов и каталогов в данном каталоге. Второй метод принимает только те элементы файловой системы, которые совпадают с заданным глобальным шаблоном.

java.nio.file.Files 7 *(окончание)*

- **static Path walkFileTree(Path *start*, FileVisitor<? super Path> *visitor*)**

  Обходит все порожденные составляющие заданного пути, применяя к ним указанный порядок обращения.

java.nio.file.SimpleFileVisitor<T> 7

- **static FileVisitResult visitFile(T *path*, BasicFileAttributes *attrs*)**

  Вызывается при обращении к файлу или каталогу. Возвращает одно из значений констант **CONTINUE**, **SKIP_SUBTREE**, **SKIP_SIBLINGS** или **TERMINATE**. В стандартной реализации по умолчанию ничего особенного не делает и только продолжает выполнение.
- **static FileVisitResult preVisitDirectory(T *dir*, BasicFileAttributes *attrs*)**
- **static FileVisitResult postVisitDirectory(T *dir*, BasicFileAttributes *attrs*)**

  Вызываются до и после обращения к каталогу. В исходной реализации по умолчанию ничего особенного не делают и только продолжают выполнение.
- **static FileVisitResult visitFileFailed(T *path*, IOException *exc*)**

  Вызывается, если генерируется исключение в связи с попыткой получить сведения о заданном файле. В стандартной реализации по умолчанию повторно генерирует исключение, прекращающее обращение к файлу с данным исключением. Этот метод следует переопределить, чтобы продолжить выполнение.

2.4.8. Системы ZIP-файлов

В классе `Paths` осуществляется поиск по путям в исходной файловой системе, т.е. там, где файлы хранятся на локальном диске пользовательского компьютера. Но ведь могут быть и другие файловые системы. К числу наиболее употребительных относится *система ZIP-файлов*. Если `zipname` — это имя ZIP-файла, то в результате следующего вызова устанавливается файловая система, содержащая все файлы в ZIP-архиве:

```
FileSystem fs = FileSystems.newFileSystem(
                Paths.get(zipname), null);
```

Если известно имя файла, его нетрудно скопировать из такого архива следующим образом:

```
Files.copy(fs.getPath(sourceName), targetPath);
```

Здесь метод `fs.getPath()` выполняет функции, аналогичные методу `Paths.get()`, для произвольной файловой системы. Чтобы перечислить все файлы в ZIP-архиве, следует обойти дерево файлов, как показано в приведенном ниже фрагменте кода. Это намного удобнее, чем пользоваться прикладным интерфейсом API, описанным в разделе 2.2.3, где требовался новый ряд классов только для обращения с ZIP-архивами.

```
FileSystem fs = FileSystems.newFileSystem(
                Paths.get(zipname), null);
Files.walkFileTree(fs.getPath("/"),
                   new SimpleFileVisitor<Path>()
  {
    public FileVisitResult visitFile(Path file,
           BasicFileAttributes attrs) throws IOException
    {
      System.out.println(file);
      return FileVisitResult.CONTINUE;
    }
  });
```

java.nio.file.FileSystems 7

- **static FileSystem newFileSystem(Path *path*, ClassLoader *loader*)**
  Обходит все установленные поставщики файловых систем, а также файловые системы, которые способен загрузить указанный загрузчик классов, если не указано пустое значение **null** параметра ***loader***. По умолчанию предоставляется поставщик для систем ZIP-файлов, принимающий файлы с расширением **.zip** или **.jar**.

java.nio.file.FileSystem 7

- **static Path getPath(String *first*, String... *more*)**
  Составляет путь, объединяя заданные символьные строки.

2.5. Файлы, отображаемые в памяти

В большинстве операционных систем можно выгодно пользоваться реализациями виртуальной памяти для отображения файла или только определенной его части в оперативной памяти. В этом случае доступ к файлу можно получить так, как будто он хранится в виде массива в оперативной памяти, что намного быстрее, чем при выполнении традиционных операций над файлами.

2.5.1. Эффективность файлов, отображаемых в памяти

В конце этого раздела приведен пример программы, вычисляющей контрольную сумму CRC32 для файла с использованием операции ввода данных из традиционного и отображаемого в памяти файла. В табл. 2.5 перечислены временные характеристики, полученные на одном компьютере при вычислении с помощью этой программы контрольной суммы для файла `rt.jar` объемом 37 Мбайт, входящего в состав комплекта JDK и расположенного в каталоге `jre/lib`.

Таблица 2.5. Временные характеристики некоторых операций над файлами

Средство выполнения операции	Время (в секундах)
Простой поток ввода	110
Буферизованный поток ввода	9.9
Файл с произвольным доступом	162
Файл, отображаемый в памяти	7.2

Как следует из табл. 2.5, на отдельно взятой машине при отображении файла в памяти потребовалось немного меньше времени, чем при последовательном вводе данных из файла с буферизацией, и значительно меньше времени, чем при произвольном доступе к файлу средствами класса `RandomAccessFile`. Разумеется, на других машинах эти показатели будут выглядеть несколько иначе, но не вызывает никаких сомнений, что выигрыш в производительности может оказаться значительным при отображении файла в памяти по сравнению с произвольным доступом к нему. В то же время для последовательного ввода из файлов среднего размера прибегать к отображению в памяти нецелесообразно.

Пакет `java.nio` делает отображение файлов в памяти довольно простым процессом. С этой целью для файла сначала получается *канал*, как показано ниже. Под каналом подразумевается предназначенная для дисковых файлов абстракция, которая позволяет получать доступ к таким функциональным возможностям операционной системы, как отображение в памяти, блокировка файлов и быстрая передача данных между файлами.

```
FileChannel channel = FileChannel.open(path, options);
```

Затем из канала получается объект типа `ByteBuffer` в результате вызова метода `map()` из класса `FileChannel`. При этом указываются отображаемая в памяти часть файла и *режим отображения*. В целом поддерживаются три следующих режима отображения.

- `FileChannel.MapMode.READ_ONLY`. Получаемый в итоге буфер служит только для чтения. Любые попытки записать данные в этот буфер приведут к исключению типа `ReadOnlyBufferException`.
- `FileChannel.MapMode.READ_WRITE`. Получаемый в итоге буфер служит как для чтения, так и для записи, благодаря чему все вносимые изменения будут в определенный момент времени записываться обратно в файл. Однако другие программы, отобразившие в памяти тот же самый файл, возможно, и не сразу обнаружат эти изменения. Конкретное поведение при одновременном отображении файла в памяти многими программами зависит от используемой операционной системы.
- `FileChannel.MapMode.PRIVATE`. Получаемый в итоге буфер служит как для чтения, так и для записи, но любые вносимые изменения относятся только к этому буферу, а следовательно, они не будут распространяться на файл.

Получив требуемый буфер, можно перейти непосредственно к чтению и записи данных с помощью методов из класса `ByteBuffer` и его суперкласса `Buffer`. Буферы поддерживают как последовательный, так и произвольный

доступ к данным. В любом буфере имеется *позиция*, продвигаемая методами `get()` и `put()`. В качестве примера ниже приведен код, требующийся для последовательного обхода всех байтов в буфере.

```
while (buffer.hasRemaining())
{
  byte b = buffer.get();
  . . .
}
```

Код, требующийся для произвольного доступа, выглядит следующим образом:

```
for (int i = 0; i < buffer.limit(); i++)
{
  byte b = buffer.get(i);
  . . .
}
```

Кроме того, читать и записывать массивы байтов можно с помощью следующих методов:

```
get(byte[] bytes)
get(byte[], int offset, int length)
```

Наконец, с помощью перечисленных ниже методов можно читать значения примитивных типов, хранящиеся в файле в *двоичном* формате.

```
getInt()      getChar()
getLong()     getFloat()
getShort()    getDouble()
```

Как упоминалось ранее, в Java для хранения данных в двоичном формате применяется обратный порядок следования байтов от старшего к младшему. Но если требуется обработать файл, содержащий данные в двоичном формате с прямым порядком следования байтов от младшего к старшему, то достаточно сделать следующий вызов:

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

Для выяснения текущего порядка следования байтов достаточно сделать вызов

```
ByteOrder b = buffer.order()
```

ВНИМАНИЕ! В двух последних методах условные обозначения имен **get/set** не соблюдаются.

Для записи числовых данных в буфер можно воспользоваться одним из перечисленных ниже методов. В какой-то момент и, конечно, тогда, когда закрывается канал, внесенные изменения записываются обратно в файл.

```
PutInt()      putChar()
putLong()     putFloat()
putShort()    putDouble()
```

В листинге 2.5 приведен пример программы, вычисляющей для файла контрольную сумму в 32-разрядном циклическом коде с избыточностью (CRC32). Такая контрольная сумма часто применяется для того, чтобы выяснить, не был

ли поврежден файл. Повреждение файла практически неизбежно ведет к изменению этой контрольной суммы. В состав пакета `java.util.zip` входит класс `CRC32`, позволяющий вычислять такую контрольную сумму для последовательности байтов в следующем цикле:

```
var crc = new CRC32();
while (есть ли дополнительные байты?)
  crc.update(следующий байт)
long checksum = crc.getValue();
```

Подробности вычисления контрольной суммы CRC32 не так важны. Но здесь оно демонстрируется лишь в качестве наглядного примера полезной операции над файлами. Запустить рассматриваемую здесь программу можно, введя следующую команду:

```
java memoryMap.MemoryMapTest имя_файла
```

Листинг 2.5. Исходный код из файла **memoryMap/MemoryMapTest.java**

```
1   package memoryMap;
2
3   import java.io.*;
4   import java.nio.*;
5   import java.nio.channels.*;
6   import java.nio.file.*;
7   import java.util.zip.*;
8
9   /**
10   * В этой программе контрольная сумма CRC32
11   * вычисляется для файла четырьмя способами
12   * Использование: java memoryMap.MemoryMapTest имя_файла
13   * @version 1.02 2018-05-01
14   * @author Cay Horstmann
15   */
16  public class MemoryMapTest
17  {
18    public static long checksumInputStream(Path filename)
19          throws IOException
20    {
21      try (InputStream in =
22           Files.newInputStream(filename))
23      {
24        var crc = new CRC32();
25
26        int c;
27        while ((c = in.read()) != -1)
28          crc.update(c);
29        return crc.getValue();
30      }
31    }
32
33    public static long
34          checksumBufferedInputStream(Path filename)
```

```
35          throws IOException
36    {
37      try (var in = new BufferedInputStream(
38           Files.newInputStream(filename)))
39      {
40        var crc = new CRC32();
41
42        int c;
43        while ((c = in.read()) != -1)
44          crc.update(c);
45        return crc.getValue();
46      }
47    }
48
49    public static long
50          checksumRandomAccessFile(Path filename)
51          throws IOException
52    {
53      try (var file =
54           new RandomAccessFile(filename.toFile(), "r"))
55      {
56        long length = file.length();
57        var crc = new CRC32();
58
59        for (long p = 0; p < length; p++)
60        {
61          file.seek(p);
62          int c = file.readByte();
63          crc.update(c);
64        }
65        return crc.getValue();
66      }
67    }
68
69    public static long
70          checksumMappedFile(Path filename)
71          throws IOException
72    {
73      try (FileChannel channel =
74           FileChannel.open(filename))
75      {
76        var crc = new CRC32();
77        int length = (int) channel.size();
78         MappedByteBuffer buffer = channel.map(
79               FileChannel.MapMode.READ_ONLY, 0, length);
80
81         for (int p = 0; p < length; p++)
82         {
83           int c = buffer.get(p);
84           crc.update(c);
85         }
86         return crc.getValue();
87      }
```

```
88    }
89
90    public static void main(String[] args)
91          throws IOException
92    {
93      System.out.println("Input Stream:");
94      long start = System.currentTimeMillis();
95      Path filename = Paths.get(args[0]);
96      long crcValue = checksumInputStream(filename);
97      long end = System.currentTimeMillis();
98      System.out.println(Long.toHexString(crcValue));
99      System.out.println((end - start)
100                       + " milliseconds");
101     System.out.println("Buffered Input Stream:");
102     start = System.currentTimeMillis();
103     crcValue = checksumBufferedInputStream(filename);
104     end = System.currentTimeMillis();
105     System.out.println(Long.toHexString(crcValue));
106     System.out.println((end - start)
107                       + " milliseconds");
108     System.out.println("Random Access File:");
109     start = System.currentTimeMillis();
110     crcValue = checksumRandomAccessFile(filename);
111     end = System.currentTimeMillis();
112     System.out.println(Long.toHexString(crcValue));
113     System.out.println((end - start)
114                       + " milliseconds");
115     System.out.println("Mapped File:");
116     start = System.currentTimeMillis();
117     crcValue = checksumMappedFile(filename);
118     end = System.currentTimeMillis();
119     System.out.println(Long.toHexString(crcValue));
120     System.out.println((end - start)
121                       + " milliseconds");
122   }
123 }
```

java.io.FileInputStream 1.0

- **FileChannel getChannel()** 1.4
  Возвращает канал для получения доступа к данному потоку ввода.

java.io.FileOutputStream 1.0

- **FileChannel getChannel()** 1.4
  Возвращает канал для получения доступа к данному потоку вывода.

java.io.RandomAccessFile 1.0

- **FileChannel getChannel()** 1.4
  Возвращает канал для получения доступа к данному файлу.

java.nio.channels.FileChannel 1.4

- **static FileChannel open(Path *path*, OpenOption... *options*)** 7
  Открывает канал доступа к файлу по заданному пути. По умолчанию канал открывается для чтения. Параметр ***options*** принимает одну из следующих констант, определяемых в перечислении **StandardOpenOption**: **WRITE**, **APPEND**, **TRUNCATE_EXISTING**, **CREATE**.
- **MappedByteBuffer map(FileChannel.MapMode *mode*, long *position*, long *size*)**
  Отображает часть файла в памяти. Параметр ***mode*** принимает одну из следующих констант, определяемых в классе **FileChannel.MapMode**: **READ_ONLY**, **READ_WRITE** или **PRIVATE**.

java.nio.Buffer 1.4

- **boolean hasRemaining()**
  Возвращает логическое значение **true**, если текущее положение в буфере еще не достигло предельной позиции.
- **int limit()**
  Возвращает предельную позицию в буфере, т.е. первую позицию, где больше нет никаких значений.

java.nio.ByteBuffer 1.4

- **byte get()**
  Получает байт из текущей позиции и продвигает текущую позицию к следующему байту.
- **byte get(int *index*)**
  Получает байт по указанному индексу.
- **ByteBuffer put(byte *b*)**
  Размещает байт на текущей позиции и продвигает ее к следующему байту. Возвращает ссылку на данный буфер.
- **ByteBuffer put(int *index*, byte *b*)**
  Размещает байт по указанному индексу. Возвращает ссылку на данный буфер.

java.nio.ByteBuffer 1.4 *(окончание)*

- **ByteBuffer get(byte[] *destination*)**
- **ByteBuffer get(byte[] *destination*, int *offset*, int *length*)**

  Заполняют байтовый массив или только какую-то его часть байтами из буфера и продвигают текущую позицию на количество считанных байтов. Если в буфере недостаточно байтов, то ничего не считывают и генерируют исключение типа **BufferUnderflowException**. Возвращают ссылку на данный буфер.
- **ByteBuffer put(byte[] *source*)**
- **ByteBuffer put(byte[]*source*, int *offset*, int *length*)**

  Размещают в буфере все байты из байтового массива или только какую-то их часть и продвигают текущую позицию на количество записанных байтов. Если в буфере слишком много байтов, то ничего не записывают и генерируют исключение типа **BufferOverflowException**. Возвращают ссылку на данный буфер.
- ***Xxx* get*Xxx*()**
- ***Xxx* get*Xxx*(int *index*)**
- **ByteBuffer put*Xxx*(*Xxx value*)**
- **ByteBuffer put*Xxx*(int *index*, *Xxx value*)**

  Получают или размещают в буфере двоичное число. Вместо обозначения ***Xxx*** может быть указан один из следующих примитивных типов данных: **Int**, **Long**, **Short**, **Char**, **Float** или **Double**.
- **ByteBuffer order(ByteOrder *order*)**
- **ByteOrder order()**

  Устанавливают или получают порядок следования байтов. Параметр ***order*** может принимать значение одной из следующих констант из класса **ByteOrder**: **BIG_ENDIAN** или **LITTLE_ENDIAN**.
- **static ByteBuffer allocate(int *capacity*)**

  Конструирует буфер заданной емкости.
- **static ByteBuffer wrap(byte[] *values*)**

  Конструирует буфер, опирающийся на заданный массив.
- **CharBuffer asCharBuffer()**

  Конструирует символьный буфер, опирающийся на данный буфер. Изменения в символьном буфере отражаются в данном буфере, но у символьного буфера имеется своя позиция, предел и отметка.

java.nio.CharBuffer 1.4

- **char get()**
- **CharBuffer get(char[] *destination*)**
- **CharBuffer get(char[] *destination*, int *offset*, int *length*)**

  Получают значение типа **char** или ряд подобных значений, начиная с указанной позиции в буфере и продвигая ее на количество считанных символов. Два последних метода возвращают ссылку **this** на данный буфер.

java.nio.CharBuffer 1.4 *(окончание)*

- **CharBuffer put(char *c*)**
- **CharBuffer put(char[] *source*)**
- **CharBuffer put(char[] *source*, int *offset*, int *length*)**
- **CharBuffer put(String *source*)**
- **CharBuffer put(CharBuffer *source*)**

  Размещают в буфере одно значение типа **char** или ряд подобных значений, начиная с указанной позиции в буфере и продвигая ее на количество записанных символов. При чтении из исходного буфера типа **CharBuffer** считываются все оставшиеся в нем символы. Возвращают ссылку **this** на данный буфер.

2.5.2. Структура буфера данных

При использовании механизма отображения файлов в памяти создается единственный буфер, охватывающий файл полностью или только интересующую его часть. Но буфера могут применяться для чтения или записи и более скромных фрагментов данных.

В этом разделе дается краткое описание основных операций, которые могут выполняться над объектами типа `Buffer`. *Буфером* называется массив значений одинакового типа. Класс `Buffer` является абстрактным с такими производными от него конкретными подклассами, как `ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer` и `ShortBuffer`.

НА ЗАМЕТКУ! Класс **StringBuffer** не имеет никакого отношения к этим подклассам, реализующим буфера данных.

На практике чаще всего применяются классы `ByteBuffer` и `CharBuffer`. Как показано на рис. 2.9, каждый буфер обладает следующим свойствами:

- *Емкость*, которая вообще не изменяется.
- *Позиция*, начиная с которой считывается или записывается следующее значение.
- *Предел*, вне которого чтение или запись не имеет смысла.
- Необязательная *отметка* для повторения операции чтения или записи.

Все эти свойства удовлетворяют следующему условию:
0 = *отметка* = *позиция* = *предел* = *емкость*

Буфер работает главным образом по циклу "сначала запись, затем чтение". Исходная позиция в буфере соответствует нулю (0), а предел — его емкости. Для ввода значений в буфер следует вызвать метод `put()`. Исчерпав вводимые в буфер данные или заполнив всю его емкость, можно переходить к чтению данных из буфера.

Чтобы установить предел на текущей позиции, а позицию — на нуле, следует вызвать метод `flip()`. Далее можно вызывать метод `get()` до тех пор, пока метод `remaining()` будет возвращать положительные значения разности *предела* и *позиции*. Прочитав все значения из буфера, следует вызвать метод `clear()`, чтобы

подготовить буфер к следующему циклу записи. Метод `clear()` устанавливает позицию в исходное нулевое положение, а предел — равным емкости буфера.

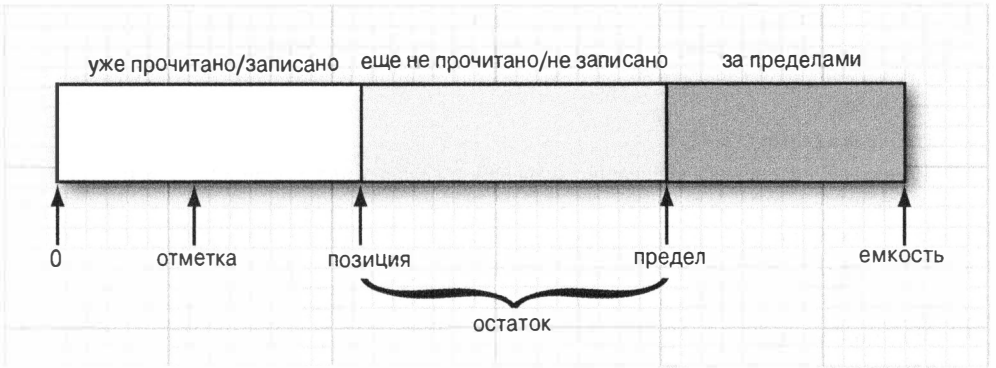


Рис. 2.9. Буфер

Если требуется выполнить повторное чтение данных из буфера, то для этого следует воспользоваться такими методами, как `rewind()` или `mark()` и `reset()`. Более подробно эти и другие методы обращения с буфером данных поясняются в приведенном далее описании прикладного интерфейса API.

Для получения самого буфера нужно вызвать статический метод `ByteBuffer.allocate()` или `ByteBuffer.wrap()`. После этого можно заполнить буфер из открытого канала или вывести его содержимое в канал. В приведенном ниже примере кода показано, как это делается. Такой способ может стать полезной альтернативой произвольному доступу к файлу.

```
ByteBuffer buffer = ByteBuffer.allocate(RECORD_SIZE);
channel.read(buffer);
channel.position(newpos);
buffer.flip();
channel.write(buffer);
```

`java.nio.Buffer` 1.4

- **`Buffer clear()`**
  Подготавливает данный буфер к записи данных, устанавливая позицию в нулевое положение, а предел — равным емкости буфера. Возвращает ссылку **`this`** на данный буфер.
- **`Buffer flip()`**
  Подготавливает данный буфер к чтению после записи, устанавливая предел на текущей позиции, а саму позицию — в нулевое положение. Возвращает ссылку **`this`** на данный буфер.
- **`Buffer rewind()`**
  Подготавливает данный буфер к повторному чтению тех же самых значений, устанавливая позицию в нулевое положение и оставляя предел неизменным. Возвращает ссылку **`this`** на данный буфер.
- **`Buffer mark()`**
  Устанавливает отметку данного буфера на текущей позиции. Возвращает ссылку **`this`** на данный буфер.

java.nio.Buffer 1.4 *(окончание)*

- **Buffer reset()**
  Устанавливает текущую позицию данного буфера на отметке, позволяя тем самым снова читать и записывать данные в буфер с отмеченной позиции. Возвращает ссылку **this** на данный буфер.
- **int remaining()**
  Возвращает оставшееся количество значений, доступных для чтения или записи в буфере, т.е. разность предела и позиции.
- **int position()**
- **void position(int *newValue*)**
  Получают или устанавливают текущую позицию в данном буфере.
- **int capacity()**
  Возвращает емкость данного буфера.

2.6.3. Блокирование файлов

Когда нескольким одновременно выполняющимся программам требуется видоизменить один и тот же файл, они должны каким-то образом взаимодействовать друг с другом, иначе они могут легко испортить файл. В качестве выхода из этого затруднительного положения могут послужить блокировки файлов. В частности, блокировка файла управляет доступом ко всему файлу или же к определенному ряду байтов в нем.

Допустим, пользовательские глобальные параметры настройки прикладной программы сохраняются в конфигурационном файле. Если пользователь вызывает два экземпляра этой программы, то вполне возможно, что в обоих ее экземплярах потребуется выполнить запись в конфигурационный файл в один и тот же момент времени. В таком случае файл должен быть заблокирован в первом экземпляре прикладной программы. Когда же во втором ее экземпляре обнаружится, что файл заблокирован, в нем может быть принято решение подождать до тех пор, пока файл не разблокируется, или вообще отказаться от записи.

Для блокирования файла можно вызывать метод `lock()` или `tryLock()` из класса `FileChannel` следующим образом:

```
FileChannel = FileChannel.open(path);
FileLock lock = channel.lock();
```

или

```
FileLock lock = channel.tryLock();
```

Когда вызывается метод `lock()`, он блокируется до тех пор, пока блокировка не будет доступна. А когда вызывается метод `tryLock()`, сразу же возвращается разрешение на блокировку или пустое значение `null`, если блокировка недоступна для установки. Файл остается заблокированным до тех пор, пока не закроется канал или не будет вызван снимающий блокировку метод `release()`.

Для блокирования только какой-нибудь определенной части файла можно сделать вызов

```
FileLock lock(long start, long size, boolean exclusive)
```

или

```
FileLock tryLock(long start, long size, boolean exclusive)
```

Если указать логическое значение `false` флага `shared`, файл будет заблокирован как для записи, так и для чтения. Если же указать логическое значение `true` этого флага, то станет доступной *разделяемая блокировка*, позволяющая нескольким процессам читать из файла, но не допускающая ни для одного из них приобретение права на исключительную блокировку. Разделяемые блокировки поддерживаются не во всех операционных системах. Поэтому вполне возможно получить право на исключительную блокировку, запрашивая только разделяемую блокировку. Чтобы выяснить, какой из этих видов блокировки доступен, следует вызывать метод `isShared()` из класса `FileLock`.

НА ЗАМЕТКУ! Если сначала блокируется концевая часть файла, а затем содержимое файла разрастается за пределы заблокированной части, то дополнительная часть файла не блокируется. Для блокирования всех байтов в файле нужно указать значение **`Long.MAX_VALUE`** его размера.

По завершении операций над файлом следует снять с него блокировку. И как всегда, это лучше всего сделать с помощью оператора `try` с ресурсами, как показано ниже.

```
try (FileLock lock = channel.lock())
{
  получить доступ к заблокированному файлу или его части
}
```

Не следует, однако, забывать, что блокирование файлов зависит от используемой системы. Ниже перечислены некоторые особенности блокирования файлов, на которые следует обращать внимание.

- В некоторых системах блокирование файлов является лишь *желательным*, но не обязательным. Если прикладной программе не удастся получить разрешение на блокировку, она все равно может начать запись данных файл, несмотря на то, что доступ к этому файлу в данный момент заблокирован другой прикладной программой.
- В некоторых системах нельзя одновременно блокировать файл и отображать его в памяти.
- Блокировки файлов удерживаются всей виртуальной машиной Java. Если с помощью одной и той же виртуальной машины запускаются сразу две программы (например, аплет или программа запуска приложений), эти программы не смогут по отдельности получать разрешение на блокировку одного и того же файла. А если виртуальная машина уже удерживает другую перекрывающую блокировку для того же самого файла, то методы `lock()` и `tryLock()` будут просто генерировать исключение типа `OverlappingFileLockException`.

- В некоторых системах закрытие канала приводит к снятию с базового файла всех блокировок, которые удерживаются виртуальной машиной Java. Поэтому лучше не открывать много каналов для одного и того же заблокированного файла.
- Блокирование файлов в сетевой файловой системе очень сильно зависит от используемой системы, и поэтому в таких системах лучше всего избегать этого механизма.

java.nio.channels.FileChannel 1.4

- **FileLock lock()**
  Получает разрешение на исключительную блокировку всего файла. Этот метод блокируется до тех пор, пока не получит разрешение на блокировку.
- **FileLock tryLock()**
  Получает разрешение на исключительную блокировку всего файла или возвращает пустое значение **null**, если блокировка не может быть получена.
- **FileLock lock(long *position*, long *size*, boolean *shared*)**
- **FileLock tryLock(long *position*, long *size*, boolean *shared*)**
  Получают разрешение на блокировку доступа к определенной части файла. Первый метод блокируется до тех пор, пока такое разрешение не будет получено, а второй возвращает пустое значение **null**, если не удается получить разрешение на блокировку.

java.nio.channels.FileLock 1.4

- **void close()** 1.7
  Снимает блокировку.

2.6. Регулярные выражения

Регулярные выражения применяются для указания шаблонов строк. С их помощью можно отыскать символьные строки, совпадающие с конкретным шаблоном. Например, в одном из рассматриваемых далее примеров программ осуществляется поиск по шаблону `<a href="...">` для обнаружения всех гиперссылок в HTML-файле.

Безусловно, для определения шаблона обозначение `...` является не достаточно точным. Необходимо как можно конкретнее указывать, какая именно последовательность символов допускается для совпадения. Поэтому для описания каждого шаблона требуется специальный синтаксис регулярных выражений. В качестве примера рассмотрим простое регулярное выражение `[Jj]ava.+`, обеспечивающее совпадение с любой символьной строкой, отвечающей следующим критериям поиска:

- начинается с буквы `J` или `j`;
- содержит `ava` на месте трех последующих букв;
- в остальной части содержит один или больше произвольных символов.

Например, строка `"javanese"` совпадает с данным регулярным выражением, а строка `"Core Java"` не совпадает.

Как видите, чтобы понять смысл регулярного выражения, нужно хотя бы немного разбираться в его синтаксисе. Правда, для большинства целей вполне хватает небольшого набора довольно простых синтаксических конструкций, рассматриваемых ниже.

- *Класс символов* — это набор альтернативных символов, заключенных в квадратные скобки, например: `[Jj]`, `[0-9]`, `[A-Za-z]` или `[^0-9]`. Здесь знаком `-` обозначается диапазон символов (т.е. все символы, значения которых в Юникоде находятся в указанных пределах), а знаком `^` — дополнение (т.е. все символы, кроме указанных).
- Чтобы включить знак `-` в класс символов, его необходимо сделать первым или последним элементом данного класса. А для того чтобы включить знак `[`, его следует сделать первым элементом. И для того чтобы включить знак `^`, его достаточно разместить где угодно, только не в начале класса символов. Экранировать необходимо только знаки `[` и `\`.
- Имеется немало предопределенных классов символов вроде **`\d`** (для цифр) или `\p{Sc}` (для знака денежной единицы в Юникоде), как показано в табл. 2.6 и 2.7.
- Большинство символов указываются для совпадения непосредственно в шаблоне, как, например, буквы `ava` в рассмотренном выше шаблоне.
- Знак `.` обозначает совпадение с любым символом, кроме символов окончания строки (в зависимости от установленных флагов).
- Знак `\` служит для экранирования символов, например, выражение `\.` обозначает совпадение с точкой, а выражение `\\` — совпадение с обратной косой чертой.
- Знаки `^` и `$` обозначают совпадение в начале и в конце строки соответственно.
- Если `X` и `Y` являются регулярными выражениями, то выражение `XY` обозначает "любое совпадение с `X`, после которого следует совпадение с `Y`", а выражение `X | Y` — "любое совпадение с `X` или `Y`".
- В выражении `X` можно применять *кванторы* вроде `X+` (`1` или больше), `X*` (`0` или больше) и `X?` (`0` или `1`).
- По умолчанию квантор обозначает совпадение с наибольшим количеством возможных повторений, определяющих удачный исход всего сопоставления с шаблоном в целом. Этот режим можно изменять с помощью суффикса `?` (обозначающего минимальное, или нестрогое, совпадение при наименьшем количестве повторений) и суффикса `+` (обозначающего максимальное, строгое или полное совпадение при наибольшем количестве

повторений, даже если это чревато неудачным исходом всего сопоставления с шаблоном в целом).

- Например, символьная строка `"cab"` совпадает с шаблоном **`[a-z]*ab`**, но не с шаблоном `[a-z]*+ab`. В первом случае с выражением `[a-z]*` совпадает только символ **`c`**, поэтому символы `ab` совпадают с остальной частью шаблона. А во втором, более строгом случае символы `cab` совпадают с выражением `[a-z]*+`, тогда как остальная часть шаблона остается не совпавшей.
- Для определения подвыражений можно использовать *группы*. Все группы следует заключать в круглые скобки, например `([+-]?)([0-9]+)`. После этого можно обратиться к сопоставителю с шаблоном, чтобы возвратить совпадение с каждой группой или обратную ссылку на отдельную группу с помощью выражения с `\n`, где `n` — номер группы, начиная с `\1`.

Таблица 2.6. Синтаксис регулярных выражений

Синтаксис	Описание	Пример
Символы		
c, любой символ, кроме знаков **`.`**, **`*`**, **`+`**, **`?`**, **`{`**, **`|`**, **`(`**, **`)`**, **`[`**, **`\`**, **`^`**, **`$`**	Символ ***c***	**`]`**
`.`	Любой символ, кроме знаков окончания строки, или любой символ, если установлен флаг **`DOTALL`**	
`\x{`*p*`}`	Кодовая точка Юникода, представленная в шестнадцатеричном коде ***p***	**`\x{1D546}`**
`\u`*hhhh*, **`\x`***hh*, **`\0`***o*, **`\0`***oo*, **`\0`***ooo*	Кодовая точка Юникода, представленная в шестнадцатеричном или восьмеричном коде	**`\uFEFF`**
`\a`, **`\e`**, **`\f`**, **`\n`**, **`\r`**, **`\t`**	Предупреждение (**`\x{7}`**), переключение кода (**`\x{1B}`**), новая строка (**`\x{A}`**), возврат каретки (**`\x{D}`**), табуляция (**`\x{9}`**)	**`\n`**
`\c`*c*, где ***c*** — буква в пределах **`[A-Z]`** или один из знаков **`@`**, **`[`**, **`\`**, **`]`**, **`^`**, **`_`**, **`?`**	Управляющий символ, соответствующий обозначению ***c***	**`\cH`** — возврат на одну позицию (**`\x{8}`**)
`\`*c*, где ***c*** — любой символ, кроме буквы или цифры в пределах **`[A-Za-z0-9]`**	Символ ***c***	**`\\`**
`\Q . . . \E`	Все, что указано от начала и до конца цитаты	Шаблон **`\Q(...)\E`** совпадает с символьной строкой `(...)`

Продолжение табл. 2.6

Синтаксис	Описание	Пример
Классы символов		
`[`C_1C_2`...]`, где C_i — символы в пределах *c-d* или классы символов	Любой символ из последовательности C_1, C_2...	`[0-9+-]`
`[^...]`	Дополнение класса символов	`[^\d\s]`
`[... && ...]`	Пересечение классов символов	`[\p{L}&&[^A-Za-z]]`
`\p{ . . . }`, `\P{ . . . }`	Предопределенный класс символов (см. табл. 2.7); его дополнение	Шаблон `\p{L` совпадает с буквой в Юникоде, как, впрочем, и шаблон `\pL`, а следовательно, фигурную скобку можно опустить
`\d`, `\D`	Цифры (по шаблону `[0-9]` или `\p{`*Цифра*`}`, если установлен флаг `UNICODE_CHARACTER_CLASS`); их дополнение	Шаблон `\d+` обозначает последовательность цифр
`\w`, `\W`	Словесные символы (по шаблону `[a-zA-Z0-9_]` или словесные символы в Юникоде, если установлен флаг `UNICODE_CHARACTER_CLASS`); их дополнение	
`\s`, `\S`	Пробелы (по шаблону `[ \n\r\t\f\x{B}]` или `\p{IsWhite_Space}`, если установлен флаг `UNICODE_CHARACTER_CLASS`); их дополнение	Шаблон `\s*, \s*` обозначает запятую, отделяемую с обеих сторон дополнительными пробелами
`\h`, `\v`, `\H`, `\V`	Горизонтальный и вертикальный пробелы, а также их дополнения	
Последовательности и альтернативы		
XY	Любая строка из выражения *X*, после которой следует любая строка из выражения *Y*	Шаблон `[1-9][0-9]*` обозначает положительное число без начального нуля
X|*Y*	Любая строка из выражения *X* или *Y*	`http|ftp`
Группирование		
`(`*X*`)`	Фиксация совпадения с выражением `X`	Шаблон `'([^']*)'` фиксирует текст в кавычках
`\`*n*	*n*-я группа	Шаблон `(['"]).*\1` совпадает со строкой `'Fred'` или `"Fred"`, но не со строкой `"Fred'`
`(?<`*имя*`>`*X*`)`	Фиксация совпадения с выражением `X` под заданным именем	Шаблон `'(?<id>[A-Za-z0-9]+)'` фиксирует совпадение с выражением под именем `id`
`\k<`*имя*`>`	Группа с заданным именем	Шаблон `\k<id>` совпадает с группой под именем `id`

Окончание табл. 2.6

Синтаксис	Описание	Пример
`(?:X)`	Употребление круглых скобок без фиксации выражения `X`	В шаблоне `(?:http|ftp)://(.*)` происходит совпадение с группой `\1` после знаков `://`
`(?`f_1f_2 `. . . :X)`, `(?`f_1 `. . . -`f_k `. . . :X)`, где f_i находится в пределах `[dimsuUx]`	Совпадение, но без фиксации с выражением `X` при установленных или сброшенных флагах (после знака `-`)	Шаблон `(?i:jpe?g)` обозначает совпадение без учета регистра
Прочее `(? . . . )`	См. ниже пояснения к классу `Pattern` в описании прикладного интерфейса API	
Кванторы		
`X?`	Необязательное наличие выражения `X`	Шаблон `\+?` обозначает присутствие необязательного знака "плюс"
`X*`, `X+`	Повторение `0`, `1` или больше раз выражения `X`	Шаблон `[1-9][0-9]+` обозначает целое число, большее или равное `10`
`X{n} X{n,} X{n,m}`	Повторение `n` раз, как минимум `n` раз, от `n` до `m` раз выражения `X`	Шаблон `[0-7]{1,3}` обозначает от одной до трех восьмеричных цифр
Q`?`, где *Q* — кванторное выражение	Принудительный квантор, пытающийся найти самое короткое совпадение, прежде чем искать более длинное совпадение	Шаблон `.*(<.+?>).*` фиксирует самую короткую последовательность, заключенную в угловые скобки
Q`+`, где *Q* — кванторное выражение	Положительный квантор, принимающий самое длинное совпадение без отката	Шаблон `'[^']*+'` совпадает со строками, заключенными в одиночные кавычки, но сразу же не совпадает со строками без закрывающей кавычки
Обнаружение границ		
`^`, `$`	Начало и конец ввода (или начало и конец строки в многострочном режиме)	Шаблон `^Java$` обозначает совпадение с введенными данными или строкой `Java`
`\A`, `\Z`, `\z`	Начало ввода, конец ввода, абсолютный конец ввода (не изменяется в многострочном режиме)	
`\b`, `\B`	Словесная граница, несловесная граница	Шаблон `\bJava\b` обозначает совпадение со словом `Java`
`\R`	Разрыв строки в Юникоде	
`\G`	Конец предыдущего совпадения	

Таблица 2.7. Имена предопределенных классов символов, применяемых с префиксом \p

Имя класса символов	Описание
posixClass	***posixClass*** — один из классов **Lower**, **Upper**, **Alpha**, **Digit**, **Alnum**, **Punct**, **Graph**, **Print**, **Cntrl**, **XDigit**, **Space**, **Blank**, **ASCII**, интерпретируемых как класс по стандарту POSIX или Unicode в зависимости от состояния флага **UNICODE_CHARACTER_CLASS**
Is*Script*, sc=*Script*, script=*Script*	Сценарий, принимаемый методом **Character.UnicodeScript.forName()**
In*Block*, blk=*Block*, block=*Block*	Блок, принимаемый методом **Character.UnicodeScript.forName()**
Category*, In*Category*, gc=*Category*, general_category=*Category	Одно- или двухбуквенное наименование общей категории символов в Юникоде
IsProperty	**Property** — одно из имен свойств Alphabetic, Ideographic, Letter, **Lowercase**, **Uppercase**, **Titlecase**, **Punctuation**, **Control**, **White_Space**, **Digit**, **Hex_Digit**, **Noncharacter_Code_Point**, **Assigned**
java*Method*	Вызов метода **Character.is*Method*()**, который не должен считаться не рекомендованным к применению

В качестве примера ниже приведено непростое, но потенциально полезное регулярное выражение (в нем описываются десятичные или шестнадцатеричные целые числа).

```
[+-]?[0-9]+|0[Xx][0-9A-Fa-f]+
```

К сожалению, синтаксис регулярных выражений не полностью стандартизирован и может выглядеть по-разному в различных программах и библиотеках, где они применяются. И хотя существует общее согласие по базовым конструкциям, тем не менее, имеется масса досадных отличий в деталях. Так, в классах, реализующих регулярные выражения в Java, применяется синтаксис регулярных выражений, подобный, но все-таки не полностью совпадающий с тем, что применяется в языке Perl. В табл. 2.6 перечислены все конструкции этого синтаксиса, внедренного в Java. Более подробные сведения о синтаксисе регулярных выражения можно найти в документации на прикладной интерфейс API для класса `Pattern` или в книге *Mastering Regular Expression, 3d Edition* Джефри Фридла (Jeffrey E. F. Friedl; издательство O'Reilly and Associates, 2006 г.[1]).

2.7.2. Совпадение со строкой

Самым простым примером применения регулярного выражения является проверка конкретной символьной строки на совпадение с ним. Ниже приведен

[1] В русском переводе книга вышла под названием *Регулярные выражения, 3-е издание*, в издательстве "Символ-Плюс", М., 2008 г.

пример кода, выполняющий такую проверку в Java. Сначала в этом коде из символьной строки, содержащей регулярное выражение, создается объект типа `Pattern`. Затем из этого объекта получается объект типа `Matcher` и вызывается его метод `matches()`.

```
Pattern pattern = Pattern.compile(patternString);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) . . .
```

В качестве входных данных для сопоставителя с шаблоном может служить объект любого класса, который реализует интерфейс `CharSequence`, например `String`, `StringBuilder` или `CharBuffer`. При компиляции шаблона можно устанавливать один или больше флагов, как показано в приведенном ниже примере кода.

```
Pattern pattern = Pattern.compile(patternString,
        Pattern.CASE_INSENSITIVE + Pattern.UNICODE_CASE);
```

С другой стороны, флаги можно указать в самом шаблоне следующим образом:

```
String regex = "(?iU:выражение)";
```

Флаги, применяемые в регулярных выражениях, перечислены ниже.

- `Pattern.CASE_INSENSITIVE` или `i`. Обозначает сопоставление символов с шаблоном без учета регистра. По умолчанию этот флаг принимает во внимание только символы в коде US ASCII.
- `Pattern.UNICODE_CASE` или `u`. В сочетании с флагом `CASE_INSENSITIVE` обозначает сопоставление символов с шаблоном, учитывая регистр букв в Юникоде.
- `Pattern.UNICODE_CHARACTER_CLASS` или `U`. Обозначает выбор классов символов по стандарту Unicode, а не POSIX. Подразумевает установку флага `UNICODE_CASE`.
- `Pattern.MULTILINE` или `m`. Обозначает применение знаков `^` и `$` для указания на сопоставление символов с шаблоном в начале и в конце строки, а не во всех входных данных.
- `Pattern.UNIX_LINES` или `d`. Обозначает распознавание только `'\n'` в качестве символа конца строки при сопоставлении символов с шаблонами `^` и `$` в многострочном режиме.
- `Pattern.DOTALL` или `s`. Обозначает совпадение со знаком `.` всех символов, включая и символы конца строки.
- `Pattern.COMMENTS` или `x`. Обозначает, что пробелы или комментарии (от знака `#` и до конца строки) игнорируются.
- `Pattern.LITERAL`. Обозначает, что шаблон воспринимается буквально и совпадение с ним должно быть точным, за исключением, возможно, регистра букв.
- `Pattern.CANON_EQ`. Обозначает необходимость учитывать каноническую эквивалентность символов юникода. Например, символ `u`, после которого следует знак `¨` (диакритический знак над гласной), будет соответствовать символу `ü`.

Два последних флага нельзя указывать в самом регулярном выражении. Если требуется найти совпадение с элементам коллекции или потока данных, соответствующий шаблон придется превратить в предикат, как показано в приведенном ниже фрагменте кода, где переменная `result` содержит все символьные строки, совпадающие с регулярным выражением.

```
Stream<String> strings = . . .;
Stream<String> result =
       strings.filter(pattern.asPredicate());
```

Если регулярное выражение содержит группы, то объект типа `Matcher` может обнаруживать их границы. Приведенные ниже методы выдают начальный и конечный индексы конкретной группы.

```
int start(int groupIndex)
int end(int groupIndex)
```

Чтобы извлечь совпавшую символьную строку, достаточно сделать следующий вызов:

```
String group(int groupIndex)
```

Под нулевой группой подразумеваются все входные данные, а индекс первой фактической группы равен `1`. Для получения сведений об общем количестве групп следует вызвать метод `groupCount()`, а для именованных групп служат перечисленные ниже методы.

```
int start(String groupName)
int end(String groupName)
String group(String groupName)
```

Вложенные группы упорядочиваются с помощью круглых скобок. Так, если используется шаблон `((1?[0-9]):([0-5][0-9]))[ap]m` и входные данные `11:59am`, то сопоставитель с шаблоном сообщит о перечисленных ниже группах.

Индекс группы	Начальная позиция	Конечная позиция	Строка
0	0	7	11:59am
1	0	5	11:59
2	0	2	11
3	3	5	59

В листинге 2.6 приведен исходный код примера программы, где сначала предлагается указать шаблон, а затем сопоставляемые с ним символьные строки, после чего сообщается, совпадают ли введенные строки с шаблоном. Если же в совпавших введенных строках и шаблоне присутствуют группы, то выводятся границы групп в круглых скобках, как показано в следующем примере:

```
((11):(59))am
```

Листинг 2.6. Исходный код из файла **`regex/RegexTest.java`**

```
1  package regex;
2
3  import java.util.*;
4  import java.util.regex.*;
```

```
5
6  /**
7   * В этой программе производится проверка на совпадение
8   * с регулярным выражением. Для этого следует ввести
9   * шаблон и сопоставляемые с ним символьные строки, а
10  * для выхода из программы — нажать клавишу пробела.
11  * Если шаблон содержит группы, их границы отображаются
12  * при совпадении
13  * @version 1.03 2018-05-01
14  * @author Cay Horstmann
15  */
16 public class RegexTest
17 {
18   public static void main(String[] args)
19         throws PatternSyntaxException
20   {
21      var in = new Scanner(System.in);
22      System.out.println("Enter pattern: ");
23      String patternString = in.nextLine();
24
25      Pattern pattern = Pattern.compile(patternString);
26
27      while (true)
28      {
29        System.out.println("Enter string to match: ");
30        String input = in.nextLine();
31        if (input == null || input.equals("")) return;
32        Matcher matcher = pattern.matcher(input);
33        if (matcher.matches())
34        {
35          System.out.println("Match");
36          int g = matcher.groupCount();
37          if (g > 0)
38          {
39            for (int i = 0; i < input.length(); i++)
40            {
41              // вывести любые пустые группы
42              for (int j = 1; j <= g; j++)
43                if (i == matcher.start(j)
44                    && i == matcher.end(j))
45                  System.out.print("()");
46              // вывести знак ( в начале непустых групп
47              for (int j = 1; j <= g; j++)
48                if (i == matcher.start(j)
49                    && i != matcher.end(j))
50                  System.out.print('(');
51              System.out.print(input.charAt(i));
52              // вывести знак ) в конце непустых групп
53              for (int j = 1; j <= g; j++)
54                if (i + 1 != matcher.start(j)
55                    && i + 1 == matcher.end(j))
56              System.out.print(')');
57            }
58            System.out.println();
59          }
60       }
```

```
61        else
62          System.out.println("No match");
63      }
64    }
65 }
```

2.7.3. Обнаружение многих совпадений

Обычно с регулярным выражением требуется сопоставлять не все входные данные, а только отыскивать в них одну или больше совпадающих символьных строк. Для поиска следующего совпадения служит метод `find()` из класса `Matcher`. Если он возвращает логическое значение `true`, то для выяснения протяженности совпадения можно далее вызвать методы `start()` и `end()`, а для получения совпавшей символьной строки — метод `group()` без аргументов, как показано ниже. Подобным способом можно обработать каждое совпадение по очереди, получив совпавшую строку и ее положение в исходной строке.

```
while (matcher.find())
{
  int start = matcher.start();
  int end = matcher.end();
  String match = input.group();
  . . .
}
```

Более изящный способ состоит в том, чтобы вызвать метод `results()` и получить в итоге поток результатов совпадения типа `Stream<MatchResult>`. В интерфейсе `MatchResult` определены такие же методы `group()`, `start()` и `end()`, как и в классе `Matcher`. (На самом деле класс `Matcher` реализует этот интерфейс.). Ниже показано, как получить список всех совпадений.

```
List<String> matches = pattern.matcher(input)
            .results()
            .map(Matcher::group)
            .collect(Collectors.toList());
```

Чтобы обнаружить совпадения в файле, можно вызвать метод `Scanner.findAll()` и получить в итоге поток результатов совпадения типа `Stream<MatchResult>`, не читая предварительно содержимое файла в символьную строку. Этому методу можно передать объект типа `Pattern` или шаблонную строку, как показано ниже.

```
var in = new Scanner(path, StandardCharsets.UTF_8);
Stream<String> words = in.findAll("\\pL+")
                         .map(MatchResult::group);
```

В листинге 2.7 приведен пример программы, где этот механизм приводится в действие. В ней отыскиваются и выводятся на экран все гипертекстовые ссылки, присутствующие на веб-странице. Чтобы запустить эту программу на выполнение, в командной строке необходимо указать какой-нибудь веб-адрес, например, следующим образом:

```
java HrefMatch http://www.horstmann.com
```

Листинг 2.7. Исходный код из файла **match/HrefMatch.java**

```
1  package match;
2
3  import java.io.*;
4  import java.net.*;
5  import java.nio.charset.*;
6  import java.util.regex.*;
7
8  /**
9   * В этой программе отображаются все веб-адреса на
10  * веб-странице путем сопоставления с регулярным
11  * выражением, описывающим дескриптор <a href=...>
12  * разметки в коде HTML.
13  * Для запуска программы следует ввести:
14  * java match.HrefMatch URL
15  * @version 1.03 2018-03-19
16  * @author Cay Horstmann
17  */
18 public class HrefMatch
19 {
20   public static void main(String[] args)
21   {
22     try
23     {
24       // извлечь символьную строку с веб-адресом (URL)
25       // из командной строки или использовать выбираемый
26       // по умолчанию URL
27       String urlString;
28       if (args.length > 0) urlString = args[0];
29       else urlString = "http://openjdk.java.net/";
30
31       // прочитать содержимое URL
32       InputStream in = new URL(urlString).openStream();
33       var input = new String(in.readAllBytes(),
34                              StandardCharsets.UTF_8);
35
36       // найти все совпадения с шаблоном
37       var patternString =
38         "<a\\s+href\\s*=\\s*(\"[^\"]*\"|[^\\s>]*)\\s*>";
39       Pattern pattern = Pattern.compile(patternString,
40                         Pattern.CASE_INSENSITIVE);
41       pattern.matcher(input)
42              .results()
43              .map(MatchResult::group)
44              .forEach(System.out::println);
45     }
46     catch (IOException | PatternSyntaxException e)
47     {
48       e.printStackTrace();
49     }
50   }
51 }
```

2.7.4. Разбиение строк по разделителям

Иногда требуется разбить исходные строки по совпадающим разделителям, чтобы извлечь их остальное содержимое. Такую задачу автоматически выполняет метод `Pattern.split()`. В итоге получается массив символьных строк с удаленными разделителями:

```
String input = . . .;
Pattern commas = Pattern.compile("\\s*,\\s*");
String[] tokens = commas.split(input);
    // исходная строка "1, 2, 3" превращается в
    // массив строк ["1", "2", "3"]
```

Если же имеется много лексем, их можно извлечь по требованию следующим образом:

```
Stream<String> tokens = commas.splitAsStream(input);
```

Если предварительная компиляция шаблона или извлечение по требованию особого значения не имеет, в таком случае достаточно вызвать лишь метод `String.split()`, как показано ниже.

```
String[] tokens = input.split("\\s*,\\s*");
```

Наконец, если исходные строки находятся в файле, в таком случае можно воспользоваться потоком сканирования, как демонстрируется в следующем примере кода:

```
var in = new Scanner(path, StandardCharsets.UTF_8);
in.useDelimiter("\\s*,\\s*");
Stream<String> tokens = in.tokens();
```

2.7.5. Замена совпадений

Метод `replaceAll()` из класса `Matcher` заменяет все совпадения символов с регулярным выражением символами из замещающей строки. Например, в приведенном ниже фрагменте кода все последовательности цифр заменяются знаком #.

```
Pattern pattern = Pattern.compile("[0-9]+");
Matcher matcher = pattern.matcher(input);
String output = matcher.replaceAll("#");
```

В замещающей строке могут содержаться ссылки на группы, присутствующие в шаблоне. Например, ссылка `$n` заменяется *n*-й группой, а ссылка `${`*имя*`}` — группой с заданным именем. Чтобы включить в текст замены знак `$`, его придется экранировать с помощью последовательности символов `\$`. Если же имеется символьная строка, содержащая знаки `$` и `\`, которые не требуется интерпретировать как замену групп, следует вызвать метод `matcher.replaceAll(Matcher.quoteReplacement(str))`. Метод `replaceFirst()` заменяет только первое совпадение с шаблоном.

Для выполнения более сложных операций, чем соединение групповых совпадений, можно предоставить замещающую функцию вместо замещающей строки. Такая функция обычно принимает объект типа `MatchResult` и возвращает

символьную строку. В качестве примера ниже демонстрируется замена всех слов, состоящих хотя бы из четырех букв, их вариантом написания прописными буквами.

```
String result = Pattern.compile("\\pL{4,}")
      .matcher("Mary had a little lamb")
      .replaceAll(m -> m.group().toUpperCase());
   // возвращается строка "MARY had a LITTLE LAMB"
```

java.util.regex.Pattern 1.4

- **static Pattern compile(String *expression*)**
- **static Pattern compile(String *expression*, int *flags*)**

  Компилируют символьную строку с регулярным выражением в объект шаблона для быстрой обработки совпадений. Параметр ***flags*** принимает одну из следующих констант: **CASE_INSENSITIVE**, **UNICODE_CASE**, **MULTILINE**, **UNIX_LINES**, **DOTALL** и **CANON_EQ**.
- **Matcher matcher(CharSequence *input*)**

  Возвращает объект типа **Matcher**, который можно использовать для обнаружения совпадений с шаблоном во входных данных.
- **String[] split(CharSequence *input*)**
- **String[] split(CharSequence *input*, int *limit*)**
- **Stream<String> splitAsStream(CharSequence *input*)** 8

  Разбивают исходную строку на лексемы, причем форму разделителей определяет шаблон. Возвращают массив или поток лексем. Разделители не являются частью лексем. Во второй форме параметр ***limit*** обозначает максимальное количество получаемых символьных строк. Так, если обнаружено ***limit*** **- 1** совпавших разделителей, последний элемент возвращаемого массива содержит оставшуюся неразбитой часть исходной строки. Если же ***limit*** **£ 0**, то разбивается вся исходная строка, А если ***limit*** **= 0**, то конечные пустые строки не вводятся в возвращаемый массив.

java.util.regex.Matcher 1.4

- **boolean matches()**

  Возвращает логическое значение **true**, если исходная строка совпадает с шаблоном.
- **boolean lookingAt()**

  Возвращает логическое значение **true**, если с шаблоном совпадает начало исходной строки.
- **boolean find()**
- **boolean find(int *start*)**

  Пытаются отыскать следующее совпадение, и если это удается, то возвращают логическое значение **true**.
- **int start()**
- **int end()**

  Возвращают начальную или следующую после конечной позицию текущего совпадения.
- **String group()**

  Возвращает текущее совпадение.

java.util.regex.Matcher 1.4 *(окончание)*

- **int groupCount()**

  Возвращает сведения о количестве групп во входном шаблоне.
- **int start(int *groupIndex*)**
- **int start(String *name*)** 8
- **int end(int *groupIndex*)**
- **int end(String *name*)** 8

  Возвращают начальную или конечную позицию данной группы в текущем совпадении. Группа обозначается индексом, начиная с **1**, **0** — для указания на полное совпадение или символьной строкой — для указания именованной группы.
- **String group(int *groupIndex*)**
- **String group((String *name*)** 7

  Возвращают символьную строку, совпадающую с заданной группой, которая обозначается индексом, начиная с **1**, **0** — для указания на полное совпадение или символьной строкой — для именования группы.
- **String replaceAll(String *replacement*)**
- **String replaceFirst(String *replacement*)**

  Возвращают символьную строку, получаемую из исходной строки в сопоставителе с шаблоном путем замены всех или только первого совпадения символами из замещающей строки. Замещающая строка может содержать ссылки **$*n*** на группы в шаблоне. Чтобы включить в нее знак **$**, следует воспользоваться последовательностью символов **\$**.
- **static String quoteReplacement(String *str*)** 5.0

  Заключает в кавычки все знаки **** и **$** в символьной строке ***str***.
- **String replaceAll(Function<MatchResult,String> *replacer*)** 9

  Заменяет каждое совпадение результатом применения функции ***replacer()*** к объекту типа **MatchResult**, содержащему результаты совпадений.
- **Stream<MatchResult> results()** 9

  Возвращает поток всех результатов совпадений.

java.util.regex.MatchResult 5

- **String group()**
- **String group(int *group*)**

  Возвращают совпавшую строку или же строку, совпавшую с заданной группой.
- **int start()**
- **int end()**
- **int start(int *group*)**
- **int end(int *group*)**

  Возвращают начальное и конечное смещение в совпавшей строке или же в строке, совпавшей в заданной группой.

java.util.Scanner 5.0

- **Stream<MatchResult> findAll(Pattern *шаблон*)** 9

  Возвращает поток всех результатов совпадений с заданным шаблоном в исходной строке, полученной из потока сканирования.

Из этой главы вы узнали, как выполняются операции ввода-вывода в Java, а также вкратце ознакомились со средствами поддержки регулярных выражений при вводе-выводе. В следующей главе речь пойдет о том, как обрабатываются данные в формате XML.

ГЛАВА 3

XML

В этой главе...

- Введение в XML
- Структура XML-документа
- Синтаксический анализ XML-документов
- Проверка достоверности XML-документов
- Поиск информации средствами XPath
- Использование пространств имен
- Потоковые синтаксические анализаторы
- Формирование XML-документов
- Преобразование XML-документов языковыми средствами XSLT

В предисловии к книге *Essential XML* Дона Бокса и др. (Don Box et al.; издательство Addison-Wesley Professional, 2000 г.) говорится, что "...расширяемый язык разметки (XML) пришел на смену языку Java, шаблонам проектирования и объектно-ориентированной технологии"... Это, конечно, шутка, но в каждой шутке есть доля правды. В самом деле, язык XML очень удобен для описания и представления структурированных данных, но он не является универсальным средством на все случаи жизни, и для его эффективного использования потребуются также специализированные по предметным областям стандарты и библиотеки. Более того, XML совсем не заменяет, а всего лишь дополняет Java. В конце 1990-х годов IBM, Apache и многие другие компании приступили к созданию на Java библиотек для обработки данных в формате XML. Наиболее важные из этих библиотек вошли в состав платформы Java.

В этой главе описаны основы языка XML, а также инструментальные средства для обработки данных в формате XML, входящие в состав библиотеки Java. Как и прежде, здесь рассматриваются случаи, когда применение XML считается

совершенно обоснованным, а также ситуации, в которых можно вполне обойтись и без этого языка, используя другие проверенные временем методики проектирования и программирования.

3.1. Введение в XML

В главе 13 первого тома настоящего издания уже приводились примеры использования *файлов свойств*, описывающих конфигурацию программы. Файл свойств содержит конфигурационные параметры в виде пар, состоящих из имени и значения, как показано ниже.

```
fontname=Times Roman
fontsize=12
windowsize=400 200
color=0 50 100
```

Для чтения такого файла единственным методом можно воспользоваться классом `Properties`. Но это отличное в целом средство не всегда подходит, поскольку во многих случаях формат файла свойств непригоден для описания данных, имеющих сложную структуру. Рассмотрим записи `fontname` и `fontsize` из приведенного выше примера. Их значения было бы удобнее объединить в одном приведенном ниже параметре, так как это в большей степени соответствовало бы объектно-ориентированному подходу.

```
font=Times Roman 12
```

Но для синтаксического анализа такого описания шрифта потребуется довольно громоздкий код, поскольку необходимо определить, где оканчивается название шрифта и начинается его размер. Дело в том, что файлы свойств имеют единую плоскую иерархию. Иногда программисты предпринимают попытки обойти данное ограничение с помощью составных имен ключей следующим образом:

```
title.fontname=Helvetica
title.fontsize=36
body.fontname=Times Roman
body.fontsize=12
```

Еще один недостаток формата файлов свойств состоит в том, что имена параметров должны быть однозначными. Для хранения последовательности значений придется употребить имена, аналогичные приведенным ниже.

```
menu.item.1=Times Roman
menu.item.2=Helvetica
menu.item.3=Goudy Old Style
```

Подобные недостатки позволяет устранить формат XML. Он служит для представления иерархических структур данных и более гибок по сравнению с плоской табличной структурой файлов свойств. Например, XML-файл с параметрами настройки программы может выглядеть следующим образом:

```
<config>
  <entry id="title">
    <font>
```

```
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </entry>
  <entry id="body">
    <font>
      <name>Times Roman</name>
      <size>12</size>
    </font>
  </entry>
  <entry id="background">
    <color>
      <red>0</red>
      <green>50</green>
      <blue>100</blue>
    </color>
  </entry>
</config>
```

Формат XML позволяет без особых затруднений выражать любую иерархическую структуру данных с повторяющимися элементами. XML-файл имеет очень простую и ясную структуру, напоминающую структуру HTML-файла. Дело в том, что оба языка, XML и HTML, созданы на основе стандартного обобщенного языка разметки SGML (Standard Generilized Markup Language).

Язык SGML в 1970-х годах использовался для описания структуры сложных документов в некоторых отраслях промышленности с высокими требованиями к документации, например, в авиастроении. Но из-за присущей ему сложности SGML так и не получил широкого распространения. Основные трудности в употреблении этого языка возникали из-за наличия двух противоречивых целей. С одной стороны, документы должны оформляться в строгом соответствии с правилами, а с другой — необходимо обеспечить простоту и высокую скорость ввода данных с помощью клавиатурных сокращений. Язык XML разработан в виде упрощенной версии SGML для Интернета. И как часто бывает в жизни, чем проще, тем лучше. Поэтому язык XML сразу же был с большим энтузиазмом воспринят теми специалистами, которые многие годы старались не употреблять SGML.

НА ЗАМЕТКУ! Очень удачно составленное описание стандарта XML можно найти по адресу **www.xml.com/axml/axml.html**.

Несмотря на общие корни, у языков XML и HTML имеется ряд существенных различий.

- В отличие от HTML, в XML учитывается регистр символов, поэтому дескрипторы `<H1>` и `<h1>` в XML считаются разными.
- В HTML некоторые закрывающие дескрипторы могут отсутствовать. Например, составитель HTML-документа может пропустить дескриптор `</p>` или `</li>`, если из контекста ясно, где заканчивается абзац или пункт списка. А в XML это не разрешается.
- Для элементов разметки без тела в XML предусмотрена сокращенная запись открывающего дескриптора, совмещенного с закрывающим. В этом

случае открывающий дескриптор заканчивается знаком /, например ****. Это означает, что наличие закрывающего дескриптора </img> подразумевается по умолчанию.

- В XML значения атрибутов должны быть заключены в кавычки, а в HTML кавычки могут отсутствовать. Например, дескриптор <applet code="MyApplet.class" width=300 height=300> можно использовать в HTML, но нельзя в XML, где значения атрибутов width и height должны быть обязательно заключены в кавычки следующим образом: width="300" и height="300".
- В HTML допускается указывать имена атрибутов без их значений, например <input type="radio" name="language" value="Java" checked>. В XML все атрибуты должны быть указаны со своими значениями, например checked="true" или checked="checked".
- Для версий 4 и 5 языка HTML в XML имеются соответствующие определения под названием XHTML.

3.2. Структура XML-документа

XML-документ должен начинаться с одного из следующих заголовков:

```
<?xml version="1.0"?>
```

или

```
<?xml version="1.0" encoding="UTF-8"?>
```

НА ЗАМЕТКУ! Язык SGML предназначался для обработки документов, поэтому XML-файлы принято называть *документами*, хотя многие XML-файлы описывают такие наборы данных, для которых этот термин не совсем подходит.

Строго говоря, указывать заголовок совсем не обязательно, но все же настоятельно рекомендуется включать его в состав документа. После заголовка обычно следует *определение типа документа* (DTD), как показано ниже.

```
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

И хотя определение DTD служит важным механизмом, обеспечивающим правильность документа, оно все же не является обязательным элементом XML-документа. Более подробно определение DTD рассматривается далее в этой главе.

Наконец, тело XML-документа содержит *корневой элемент*, который может состоять из других элементов:

```
<?xml version="1.0"?>
<!DOCTYPE config . . .>
<config>
  <entry id="title">
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
```

```
  </entry>
  . . .
</config>
```

Каждый элемент разметки может содержать *дочерние элементы*, текст или и то и другое. В приведенном выше примере элемент разметки `font` состоит из двух дочерних элементов, `name` и `size`, причем элемент `name` содержит текст `"Helvetica"`.

СОВЕТ. XML-документы рекомендуется составлять таким образом, чтобы элементы разметки содержали одно из двух: дочерние элементы или текст. Иначе говоря, следует избегать разметки, аналогичной приведенной ниже.

```
<font>
  Helvetica
  <size>36</size>
</font>
```

В спецификации XML такая разметка называется *смешанным содержимым*. Как станет ясно в дальнейшем, синтаксический анализ XML-документа намного упрощается, если избегать в нем смешанного содержимого.

Элементы разметки XML-документов могут содержать атрибуты, как показано ниже.

```
<size unit="pt">36</size>
```

Среди разработчиков XML нет единого мнения, когда следует употреблять элементы, а когда — атрибуты. Например, описать шрифт, по-видимому, проще следующим образом:

```
<font name="Helvetica" size="36"/>
```

чем так, как показано ниже.

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

Но в то же время атрибуты намного менее удобны. Допустим, в определение размера шрифта требуется добавить единицу измерения. Если воспользоваться для этой цели атрибутами, то единицы измерения придется указать рядом со значением атрибута следующим образом:

```
<font name="Helvetica" size="36 pt"/>
```

Но тогда придется написать дополнительный код для синтаксического анализа символьной строки `"36 pt"`, а именно этого стремились избежать создатели XML. Поэтому более простым решением было бы применение атрибута в элементе `size`, как показано ниже.

```
<font>
  <name>Helvetica</name>
  <size unit="pt">36</size>
</font>
```

Широко распространенное эмпирическое правило гласит: атрибуты следует использовать не для указания значений, а только при изменении их интерпретации. Если же непонятно, обозначает ли какой-нибудь атрибут изменение интерпретации значения или нет, то лучше отказаться от атрибута и употребить элемент разметки. Во многих удобно составленных XML-документах атрибуты вообще не употребляются.

НА ЗАМЕТКУ! В языке HTML существует очень простое правило употребления атрибутов: если данные не отображаются на веб-странице, значит, это атрибут. Рассмотрим следующую гипертекстовую ссылку:

```
<a href="http://java.sun.com">Java Technology</a>
```

Строка **"Java Technology"** отображается на веб-странице, но URL этой гипертекстовой ссылки не выводится. Впрочем, это правило не совсем подходит для XML-файлов, поскольку данные в XML-файле не всегда предназначены непосредственно для просмотра в удобочитаемом виде.

Элементы разметки и текст являются основными составляющими XML-документов, но в них можно также встретить и ряд других инструкций разметки.

- *Ссылки на символы* в виде `&#десятичное_значение;` или `&#xшестнадцатеричное_значение;`. Например, ссылка `&#233;` или `&#xD9;` обозначает символ é.
- *Ссылки на сущности* в виде `&имя;`. Так, ссылки на сущности `<`, `>`, `&`, `&quot;`, `&apos;` имеют предопределенные значения и соответствуют знакам <, >, &, " и '. В определении DTD можно также указать другие ссылки на сущности.
- *Разделы CDATA*, разграничиваемые последовательностями символов `<![CDATA[` и `]]>`. Они предназначены для включения строк со знаками <, > или &, которые не следует интерпретировать как символы разметки, например:

  ```
  <![CDATA[< & > мои излюбленные разделители]]>
  ```

  В разделах CDATA не допускается наличие символьных строк вроде "]]>", поэтому пользоваться ими следует очень внимательно! Зачастую они выполняют функции своего рода "лазейки" для внедрения в XML-документ данных в устаревшем формате.
- *Инструкции обработки* — это инструкции для прикладных программ, обрабатывающих XML-документы. Такие инструкции разграничиваются знаками <? и ?>, как в приведенном ниже примере.

  ```
  <?xml-stylesheet href="mystyle.css" type="text/css"?>
  ```

  Каждый XML-документ начинается со следующей инструкции обработки:

  ```
  <?xml version="1.0"?>
  ```
- *Комментарии* разграничиваются знаками `<!--` и `-->` следующим образом:

  ```
  <!-- Это комментарий. -->
  ```

В комментариях не допускаются символьные строки вроде "--". Комментарии предназначены для пользователей, поэтому в них не следует вводить скрытые команды. Для выполнения команд предназначены инструкции обработки.

3.3. Синтаксический анализ XML-документов

Для обработки XML-документа необходимо выполнить его *синтаксический анализ*. *Синтаксическим анализатором* называется такая программа, которая считывает файл, подтверждает правильность его формата, разбивает данные на составные элементы и предоставляет программисту доступ к ним. Ниже приведены две основные разновидности XML-анализаторов.

- Древовидные анализаторы, которые считывают XML-документ и представляют его в виде древовидной структуры (например, анализатор объектной модели документа, сокращенно называемый DOM-анализатором).
- Потоковые анализаторы, которые генерируют события по мере чтения XML-документа (например, простые анализаторы прикладного интерфейса API для XML, сокращенно называемые SAX-анализаторами).

DOM-анализатор проще в употреблении, поэтому сначала рассматривается именно он. Потоковый анализатор обычно применяется для обработки длинных документов, когда для древовидного представления XML-данных требуется большой объем памяти. Кроме того, его можно употреблять для извлечения отдельных элементов XML-документа без учета контекста. Подробнее об этом — в разделе 3.7.

Интерфейс DOM-анализатора стандартизован консорциумом W3C. Так, пакет `org.w3c.dom` содержит определения типов интерфейсов, в том числе `Document` и `Element`. Различные поставщики, среди которых компании IBM и Apache, разработали собственные варианты DOM-анализаторов, реализующие эти интерфейсы. В прикладном интерфейсе API для обработки XML-документов на Java, сокращенно называемом библиотекой JAXP, предусмотрена возможность подключения таких анализаторов. Кроме того, в состав комплекта JDK входит собственный DOM-анализатор. Именно он и рассматривается далее в этой главе.

Для чтения XML-документа сначала потребуется объект типа `DocumentBuilder`, который можно получить из класса `DocumentBuilderFactory` следующим образом:

```
DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

Затем можно приступать к чтению данных из файла, как показано ниже.

```
File f = . . .
Document doc = builder.parse(f);
```

С другой стороны, XML-документ можно прочитать и по указанному URL, как демонстрируется в следующем примере кода:

```
URL u = . . .
Document doc = builder.parse(u);
```

Для чтения XML-документа можно даже указать произвольный поток ввода следующим образом:

```
InputStream in = . . .
Document doc = builder.parse(in);
```

НА ЗАМЕТКУ! Если в качестве источника данных служит произвольный поток ввода, синтаксический анализатор не сможет найти те файлы, расположение которых указано относительно данного документа, например, DTD-файл, находящийся в том же каталоге. Для преодоления этого препятствия достаточно установить так называемый "определитель сущностей". Подробнее об этом можно узнать по адресу **www.xml.com/pub/a/2004/03/03/catalogs.html** или **www.ibm.com/developerworks/xml/library/x-mxd3.html**.

Объект типа `Document` является внутренним представлением древовидной структуры XML-документа. Он состоит из экземпляров классов, реализующих интерфейс `Node` и различные интерфейсы, производные от него. На рис. 3.1 показана иерархия наследования интерфейса `Node`.

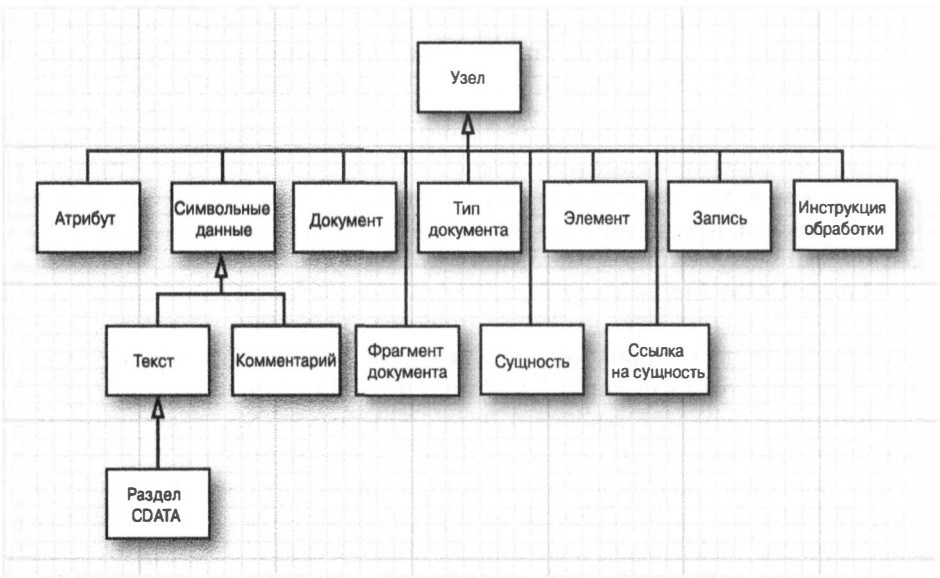


Рис. 3.1. Иерархия наследования интерфейса **Node**

Анализ содержимого документа начинается с вызова метода `getDocumentElement()`, который возвращает корневой элемент, как показано ниже.

```
Element root = doc.getDocumentElement();
```

Так, если обрабатывается приведенный ниже XML-документ, то в результате вызова метода `getDocumentElement()` будет получен элемент разметки `font`.

```
<?xml version="1.0"?>
<font>
  . . .
</font>
```

Метод `getTagName()` возвращает имя дескриптора элемента разметки. Так, если обратиться к приведенному выше примеру, то в результате вызова `root.getTagName()` возвращается символьная строка `"font"`.

Для извлечения элементов, дочерних по отношению к данному (ими могут быть подчиненные элементы, текст, комментарии или другие узлы), служит метод `getChildNodes()`, возвращающий набор данных типа `NodeList`. Этот тип данных существовал еще до создания стандартной библиотеки коллекций в Java, и поэтому для него имеется другой протокол доступа. Метод `item()` возвращает элемент набора данных по указанному индексу, а метод `getLength()` — общее количество элементов. Таким образом, для перечисления всех дочерних элементов можно воспользоваться следующим кодом:

```
NodeList children = root.getChildNodes();
for (int i = 0; i < children.getLength(); i++)
{
  Node child = children.item(i);
  . . .
}
```

Анализ дочерних элементов следует выполнять очень внимательно. На первый взгляд, приведенный ниже XML-документ содержит два элемента, дочерних по отношению к элементу разметки `font`.

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

Но синтаксический анализатор сообщит, что в разметке данного XML-документа имеется пять дочерних элементов.

- Разделитель в виде пробела между дескрипторами `<font>` и `<name>`.
- Элемент `name`.
- Разделитель в виде пробела между дескрипторами `</name>` и `<size>`.
- Элемент `size`.
- Разделитель в виде пробела между дескрипторами `</size>` и `</font>`.

На рис. 3.2 схематически представлено дерево DOM — объектной модели упомянутого выше документа.

Если требуется обработать только подчиненные элементы, в таком случае можно пренебречь всеми разделителями в виде пробелов. Это можно сделать с помощью приведенного ниже кода. В итоге будут выявлены только элементы с именами дескрипторов `name` и `size`.

```
for (int i = 0; i < children.getLength(); i++)
{
  Node child = children.item(i);
  if (child instanceof Element)
  {
    Element childElement = (Element) child;
    . . .
  }
}
```

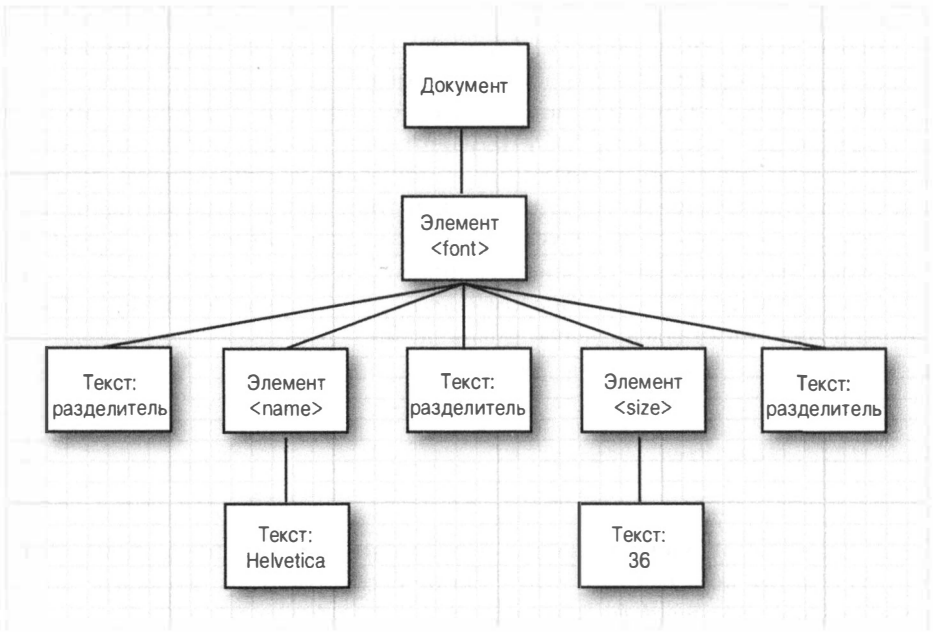


Рис. 3.2. Простое дерево DOM

Как будет показано в следующем разделе, данную задачу можно решить еще лучше, если воспользоваться определением DTD. В таком случае синтаксическому анализатору будет известно, у каких именно элементов отсутствуют текстовые узлы в качестве дочерних элементов. Благодаря этому он может подавить разделители автоматически.

При анализе элементов `name` и `size` придется извлечь содержащиеся в них текстовые строки, которые находятся в дочерних узлах типа `Text`. А поскольку заранее известно, что другие дочерние узлы отсутствуют, то можно вызвать метод `getFirstChild()` без перебора содержимого очередной коллекции типа `NodeList`. После этого с помощью метода `getData()` можно извлечь текстовую строку из узла типа `Text`, как показано в приведенном ниже фрагменте кода.

```
for (int i = 0; i < children.getLength(); i++)
{
  Node child = children.item(i);
  if (child instanceof Element)
  {
    var childElement = (Element) child;
    var textNode = (Text) childElement.getFirstChild();
    String text = textNode.getData().trim();
    if (childElement.getTagName().equals("name"))
      name = text;
    else if (childElement.getTagName().equals("size"))
      size = Integer.parseInt(text);
  }
}
```

СОВЕТ. По значению, возвращаемому в результате выполнения метода **getData()**, рекомендуется вызвать метод **trim()**. Допустим, составитель XML-документа разместил открывающие и закрывающие дескрипторы в отдельных строках, как показано ниже.

```
<size>
  36
</size>
```

В таком случае синтаксический анализатор включит все пробелы и символы перевода строк в данные из текстового узла, а метод **trim()** удалит их и оставит лишь конкретные данные.

Кроме того, для извлечения последнего дочернего узла можно воспользоваться методом `getLastChild()`, а для получения следующего родственного узла — методом `getNextSibling()`. С помощью этих методов можно обойти дочерние узлы другим способом, как показано ниже.

```
for (Node childNode = element.getFirstChild();
  childNode != null;
  childNode = childNode.getNextSibling())
{
  . . .
}
```

Для перечисления атрибутов узла следует вызвать метод `getAttributes()`, возвращающий объект типа `NamedNodeMap`, который содержит объекты типа `Node`, описывающие атрибуты. Обход узлов в именованном отображении типа `NamedNodeMap` можно выполнить таким же образом, как и обход узлов в списке типа `NodeList`. В таком случае для извлечения имен атрибутов и их значений следует воспользоваться методами `getNodeName()` и `getNodeValue()`, как показано в приведенном ниже фрагменте кода.

```
NamedNodeMap attributes = element.getAttributes();
for (int i = 0; i < attributes.getLength(); i++)
{
  Node attribute = attributes.item(i);
  String name = attribute.getNodeName();
  String value = attribute.getNodeValue();
  . . .
}
```

С другой стороны, если известно имя атрибута, его значение можно извлечь непосредственно следующим образом:

```
String unit = element.getAttribute("unit");
```

Описанный выше способ анализа дерева DOM демонстрируется в примере программы из листинга 3.1, где XML-документ преобразуется в формат JSON. В данной древовидной структуре представлены дочерние узлы, окруженные разделителями в виде пробелов и комментариями. Для большей наглядности программа отображает все символы перевода строки и возврата каретки в виде последовательности `\n`.

Чтобы понять, каким образом в данной программе осуществляется анализ дерева DOM, совсем не обязательно знать особенности формата JSON. Достаточно обратить внимание на следующее.

- Для чтения объекта типа `Document` из файла применяется объект типа `DocumentBuilder`.
- Для каждого элемента разметки выводится имя дескриптора, атрибуты и подчиненные элементы.
- Для символьных данных получается строка с данными. Если данные поступают из комментария, то добавляется префикс `"Comment: "`.

Листинг 3.1. Исходный код из файла **dom/TreeViewer.java**

```
1  package dom;
2
3  import java.io.*;
4  import java.util.*;
5
6  import javax.xml.parsers.*;
7
8  import org.w3c.dom.*;
9  import org.w3c.dom.CharacterData;
10 import org.xml.sax.*;
11
12 /**
13  * В этой программе XML-документ отображается
14  * как дерево в формате JSON
15  * @version 1.2 2018-04-02
16  * @author Cay Horstmann
17  */
18 public class JSONConverter
19 {
20   public static void main(String[] args)
21         throws SAXException, IOException,
22                ParserConfigurationException
23   {
24     String filename;
25     if (args.length == 0)
26     {
27       try (var in = new Scanner(System.in))
28       {
29         System.out.print("Input file: ");
30         filename = in.nextLine();
31       }
32     }
33     else
34       filename = args[0];
35     DocumentBuilderFactory factory =
36             DocumentBuilderFactory.newInstance();
37     DocumentBuilder builder =
38             factory.newDocumentBuilder();
39
40     Document doc = builder.parse(filename);
41     Element root = doc.getDocumentElement();
42     System.out.println(convert(root, 0));
43   }
```

```
44
45    public static StringBuilder convert(
46                  Node node, int level)
47    {
48      if (node instanceof Element)
49      {
50        return elementObject((Element) node, level);
51      }
52      else if (node instanceof CharacterData)
53      {
54        return characterString(
55               (CharacterData) node, level);
56      }
57      else
58      {
59        return pad(new StringBuilder(), level)
60              .append(jsonEscape(node.getClass()
61              .getName()));
62      }
63    }
64
65    private static Map<Character, String> replacements =
66                  Map.of('\b', "\\b", '\f', "\\f", '\n',
67                         "\\n", '\r', "\\r", '\t', "\\t",
68                         '"', "\\\"", '\\', "\\\\");
69
70    private static StringBuilder jsonEscape(String str)
71    {
72      var result = new StringBuilder("\"");
73      for (int i = 0; i < str.length(); i++)
74      {
75        char ch = str.charAt(i);
76        String replacement = replacements.get(ch);
77        if (replacement == null) result.append(ch);
78        else result.append(replacement);
79      }
80      result.append("\"");
81      return result;
82    }
83
84    private static StringBuilder
85          characterString(CharacterData node, int level)
86    {
87      var result = new StringBuilder();
88      StringBuilder data = jsonEscape(node.getData());
89      if (node instanceof Comment)
90        data.insert(1, "Comment: ");
91      pad(result, level).append(data);
92      return result;
93    }
94
95    private static StringBuilder
96            elementObject(Element elem, int level)
97    {
```

```
98      var result = new StringBuilder();
99      pad(result, level).append("{\n");
100     pad(result, level + 1).append("\"name\": ");
101     result.append(jsonEscape(elem.getTagName()));
102     NamedNodeMap attrs = elem.getAttributes();
103     if (attrs.getLength() > 0)
104     {
105       pad(result.append(",\n"), level + 1)
106                 .append("\"attributes\": ");
107       result.append(attributeObject(attrs));
108     }
109     NodeList children = elem.getChildNodes();
110     if (children.getLength() > 0)
111     {
112       pad(result.append(",\n"), level + 1)
113                 .append("\"children\": [\n");
114       for (int i = 0; i < children.getLength(); i++)
115       {
116         if (i > 0) result.append(",\n");
117         result.append(convert(children.item(i),
118                               level + 2));
119       }
120       result.append("\n");
121       pad(result, level + 1).append("]\n");
122     }
123     pad(result, level).append("}");
124     return result;
125   }
126
127   private static StringBuilder
128           pad(StringBuilder builder, int level)
129   {
130     for (int i = 0; i < level; i++)
131       builder.append(" ");
132     return builder;
133   }
134
135   private static StringBuilder
136           attributeObject(NamedNodeMap attrs)
137   {
138     var result = new StringBuilder("{");
139     for (int i = 0; i < attrs.getLength(); i++)
140     {
141       if (i > 0) result.append(", ");
142       result.append(jsonEscape(attrs.item(i)
143             .getNodeName()));
144       result.append(": ");
145       result.append(jsonEscape(attrs.item(i)
146             .getNodeValue()));
147     }
148     result.append("}");
149     return result;
150   }
151 }
```

javax.xml.parsers.DocumentBuilderFactory 1.4

- **static DocumentBuilderFactory newInstance()**
  Возвращает экземпляр класса **DocumentBuilderFactory**.
- **DocumentBuilder newDocumentBuilder()**
  Возвращает экземпляр класса **DocumentBuilder**.

javax.xml.parsers.DocumentBuilder 1.4

- **Document parse(File *f*)**
- **Document parse(String *url*)**
- **Document parse(InputStream *in*)**
  Выполняют синтаксический анализ XML-документа, полученного из заданного файла, по указанному URL или из заданного потока ввода. Возвращают результат синтаксического анализа.

org.w3c.dom.Document 1.4

- **Element getDocumentElement()**
  Возвращает корневой элемент разметки документа.

org.w3c.dom.Element 1.4

- **String getTagName()**
  Возвращает имя элемента разметки.
- **String getAttribute(String *name*)**
  Возвращает значение атрибута с заданным именем или пустую символьную строку, если такой атрибут отсутствует.

org.w3c.dom.Node 1.4

- **NodeList getChildNodes()**
  Возвращает список, содержащий все дочерние узлы данного узла.
- **Node getFirstChild()**
- **Node getLastChild()**
  Возвращают первый или последний дочерний узел данного узла. Если у данного узла отсутствуют дочерние узлы, возвращается пустое значение **null**.
- **Node getNextSibling()**
- **Node getPreviousSibling()**
  Возвращают предыдущий родственный узел. Если у данного узла отсутствуют родственные узлы, возвращается пустое значение **null**.

org.w3c.dom.Node 1.4 (окончание)

- **Node getParentNode()**
  Возвращает родительский узел данного узла или пустое значение **null**, если данный узел является узлом документа.
- **NamedNodeMap getAttributes()**
  Возвращает отображение узлов, содержащее узлы типа **Attr** с описаниями всех атрибутов данного узла.
- **String getNodeName()**
  Возвращает имя данного узла. Если узел относится к типу **Attr**, то возвращается имя атрибута.
- **String getNodeValue()**
  Возвращает значение данного узла. Если узел относится к типу **Attr**, то возвращается значение атрибута.

org.w3c.dom.CharacterData 1.4

- **String getData()**
  Возвращает текст, хранящийся в данном узле.

org.w3c.dom.NodeList 1.4

- **int getLength()**
  Возвращает количество узлов в данном списке.
- **Node item(int *index*)**
  Возвращает узел с заданным индексом. Значение индекса может быть от 0 до **getLength() - 1**.

org.w3c.dom.NamedNodeMap 1.4

- **int getLength()**
  Возвращает количество узлов в данном отображении.
- **Node item(int *index*)**
  Возвращает узел с заданным индексом. Значение индекса может быть от 0 до **getLength() - 1**.

3.4. Проверка достоверности XML-документов

В предыдущем разделе был описан способ обхода древовидной структуры DOM-документа. Но если следовать этому способу непосредственно, то потребуется приложить немало усилий для проверки ошибок программным путем. В этом случае придется не только организовать поиск и удаление лишних разделителей между элементами, но и проверить, содержит ли документ

предполагаемые узлы. Рассмотрим в качестве примера следующий элемент разметки:

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

При чтении первого же дочернего узла неожиданно обнаруживается, что это текстовый узел, содержащий разделитель "\n". Пропуская текстовые узлы, нетрудно дойти до узла первого элемента, где необходимо проверить, имеет ли его дескриптор имя `name`. Затем требуется выяснить, имеет ли он дочерний узел типа `Text`. После этого можно переместиться к следующему дочернему узлу без разделителя в виде пробела и выполнить такую же проверку. Но что делать, если составитель XML-документа изменит порядок расположения дочерних узлов или добавит еще один дочерний элемент? С одной стороны, для проверки всех возможных ошибок придется написать очень громоздкий код, а с другой — исключить такую проверку было бы слишком опрометчиво.

Правда, к числу главных преимуществ XML-анализатора относится его способность автоматически проверять корректность структуры документа. В таком случае анализ XML-документа значительно упрощается. Так, если известно, что элемент разметки `font` успешно прошел проверку, то несложно получить два дочерних узла, привести их к типу `Text`, а затем извлечь текстовые данные без дополнительной проверки.

Для указания структуры документа можно предоставить определение DTD или XML Schema. Определение DTD или XML Schema содержит правила, регламентирующие структуру документа. Оно задает допустимые дочерние узлы элементов и атрибутов каждого элемента. Например, определение DTD может содержать следующее правило:

```
<!ELEMENT font (name,size)>
```

Это правило выражает следующее ограничение: у элемента разметки `font` всегда должны быть два дочерних узла: `name` и `size`. На языке XML Schema то же самое ограничение записывается следующим образом:

```
<xsd:element name="font">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
  </xsd:sequence>
</xsd:element>
```

Язык XML Schema позволяет формулировать более изощренные условия проверки достоверности, чем определения DTD. Например, элемент разметки `size` должен содержать целочисленное значение. В отличие от определения DTD, в XML Schema используется синтаксис XML, что упрощает обработку файлов со схемами XML-документов.

В следующем разделе подробно обсуждаются определения DTD, а затем вкратце рассматриваются основные средства поддержки XML Schema. После этого будет представлен пример, наглядно демонстрирующий, насколько проверка достоверности XML-документов упрощает их обработку.

3.4.1. Определения типов документов

Существует несколько способов предоставить определение типа документа(DTD). В частности, определение DTD можно ввести в начале XML-документа:

```
<?xml version="1.0"?>
<!DOCTYPE config [
  <!ELEMENT config . . .>
  другие правила
  . . .
]>
<config>
  . . .
</config>
```

Как видите, эти правила заключаются в квадратные скобки объявления DOCTYPE. Тип документа должен соответствовать имени корневого элемента (в данном примере — config). Размещать определения DTD в самом документе неудобно, поскольку они могут быть очень длинными. Следовательно, определения DTD имеет смысл хранить в отдельном файле. А для связывания определений DTD с XML-документами можно воспользоваться приведенными ниже объявлениями SYSTEM, где указываются URL для доступа к внешним файлам конфигурации с определениями DTD.

```
<!DOCTYPE config SYSTEM "config.dtd">
```

или

```
<!DOCTYPE config SYSTEM
          "http://myserver.com/config.dtd">
```

ВНИМАНИЕ! Если для указания внешнего файла конфигурации с определением DTD служит относительный URL (например, **"config.dtd"**), то вместо потока ввода типа **InputStream** синтаксическому анализатору следует предоставить объект типа **File** или **URL**. Если же требуется синтаксический анализ данных из потока ввода, то синтаксическому анализатору следует предоставить определитель сущностей, как поясняется в следующем далее примечании.

Механизм обозначения хорошо известных определений DTD унаследован из языка SGML и демонстрируется в приведенном ниже примере. Если XML-процессору известен способ обнаружения DTD с помощью идентификатора PUBLIC, то обращаться по указанному URL совсем не обязательно.

```
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems,
          Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

НА ЗАМЕТКУ! Системный идентификатор URL определения DTD может фактически оказаться неработоспособным или намеренно действующим замедленно. Примером последнего служит системный идентификатор строгого определения DTD в версии XHTML 1.0 (https://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd). Если произвести синтаксический анализ XHTML-файла, то на обслуживание определения DTD может уйти одна или две минуты.

В качестве выхода из данного положения можно воспользоваться *определителем сущностей*, отображающим открытые идентификаторы на локальные файлы. До версии Java 9

с этой целью приходилось предоставлять объект класса, реализующего интерфейс **EntityResolver** и метод **resolveEntity()**.

В настоящее время для организации подобного отображения можно воспользоваться *каталогами XML*. Сначала предоставляется один или несколько *файлов каталогов* в следующей форме:

```
<?xml version="1.0"?>
<!DOCTYPE catalog PUBLIC
          "-//OASIS//DTD XML Catalogs V1.0//EN"
          "http://www.oasis-open.org/committees/
                  entity/release/1.0/catalog.dtd">
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
               prefer="public">
  <public publicId=". . ." uri=". . ."/>
  . . .
</catalog>
```

Затем конструируется и устанавливается определитель сущностей, как показано ниже. Полный пример приведен в листинге 3.6.

```
builder.setEntityResolver(CatalogManager.catalogResolver(
  CatalogFeatures.defaults(),
  Paths.get("catalog.xml").toAbsolutePath().toUri()));
```

Вместо того чтобы задавать местоположение файлов каталогов в прикладной программе, их можно указать в командной строке с помощью системного свойства **javax.xml.catalog.files**, введя абсолютные URL в формате **file** через точку с запятой.

Рассмотрим теперь различные правила, которые могут задаваться в определении DTD. Правило ELEMENT задает дочерние узлы данного элемента в виде регулярного выражения, составляющие которого перечислены в табл. 3.1.

Таблица 3.1. Правила для содержимого документа

Правило	Назначение
E*	0 или больше вхождений элемента ***E***
***E**+*	1 или больше вхождений элемента ***E***
***E**?*	0 или 1 вхождение элемента ***E***
$E_1 \mid E_2 \mid \ldots \mid E_n$	Один из элементов $E_1, E_2, \ldots E_n$
$E_1, E_2, \ldots E_n$	Элемент E_i, после которого следуют элементы $E_2, \ldots E_n$
#PCDATA	Текст
(#PCDATA$\mid E_1 \mid E_2 \mid \ldots \mid E_n$**)***	0 или больше вхождений текста и последовательность элементов $E_1, E_2, \ldots E_n$ в любом порядке (смешанное содержимое)
ANY	Любой дочерний узел
EMPTY	Дочерние узлы отсутствуют

Рассмотрим несколько простых примеров. Следующее правило указывает на то, что элемент разметки menu может содержать 0 или больше элементов **item**:

```
<!ELEMENT menu (item)*>
```

По приведенным ниже правилам шрифт описывается именем, после которого следует размер шрифта. Имя и размер шрифта являются текстовыми элементами.

```
<!ELEMENT font (name,size)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT size (#PCDATA)>
```

Сокращение PCDATA обозначает проанализированные символьные данные. Данные называются *проанализированными*, поскольку синтаксический анализатор обрабатывает текстовую строку и ищет знак <, обозначающий начало нового дескриптора, или же знак &, обозначающий начало сущности. Спецификация элемента может содержать регулярные выражения, в том числе вложенные и сложные. В качестве примера ниже приведено правило, описывающее структуру главы в книге. Каждая глава (chapter) начинается с введения (intro), после которого следует один или несколько разделов, состоящих из заголовка (heading), одного или нескольких абзацев (para), рисунков (image), таблиц (table) или примечаний (note).

```
<!ELEMENT chapter (intro,(heading,
                  (para|image|table|note)+)+)
```

Но правила далеко не всегда обеспечивают достаточную гибкость. Очень часто элементы содержат текст, и тогда допускаются только два варианта. Во-первых, в составе элемента допускается наличие только текста:

```
<!ELEMENT name (#PCDATA)>
```

Во-вторых, элемент может содержать *любое* сочетание *текста* и *дескрипторов*, располагаемых в произвольном порядке:

```
<!ELEMENT para (#PCDATA|em|strong|code)*>
```

Другие правила типа #PCDATA не допускаются. Например, приведенное ниже выражение неверно. Такие правила следует переписать заново, введя еще один элемент разметки caption, содержащий надпись, или допустив любое сочетание элементов разметки image и текста.

```
<!ELEMENT captionedImage (image,#PCDATA)>
```

Подобное ограничение упрощает работу XML-анализатора при синтаксическом анализе *смешанного содержимого* (текста и дескрипторов). Но в этом случае содержимое становится неконтролируемым, поэтому рекомендуется составлять такие определения DTD, которые содержат только элементы разметки или же ничего, кроме текста.

НА ЗАМЕТКУ! Не совсем верно считать, что в правилах DTD можно указывать произвольные регулярные выражения. XML-анализатор может отклонить сложные наборы правил, которые до определенного момента не дают однозначного результата. Примером тому служит регулярное выражение **((x,y)|(x,z))**. Обнаружив в нем элемент **x**, анализатор не сможет сразу выяснить, какой из двух альтернативных вариантов выбрать. Это выражение следует переписать в следующем виде: **(x,(y|z))**. Но некоторые определения нельзя изменить, например **((x,y)*x?)**. Синтаксический анализатор из библиотеки Java XML не выдает никаких предупреждений при обнаружении подобных определений DTD. Он просто выбирает первый совпадающий вариант, что может привести к неверной интерпретации правильных входных данных.

Для описания допустимых атрибутов элементов разметки используется приведенный ниже синтаксис. В табл. 3.2 перечислены допустимые типы атрибутов, а в табл. 3.3 — синтаксис поведения атрибутов по умолчанию.

```
<!ATTLIST элемент атрибут тип поведение_по_умолчанию>
```

Таблица 3.2. Типы атрибутов

Тип	Назначение
CDATA	Произвольная символьная строка
(A_1|A_2| ... |A_n)	Один из строковых атрибутов **A_1**, **A_2**, ... **A_n**
NMTOKEN, NMTOKENS	Одна или несколько лексем, соответствующих имени
ID	Однозначный идентификатор
IDREF, IDREFS	Одна или несколько ссылок на однозначный идентификатор
ENTITY, ENTITIES	Одна или несколько непроанализированных сущностей

Таблица 3.3. Поведение атрибутов по умолчанию

Поведение по умолчанию	Назначение
#REQUIRED	Атрибут является обязательным
#IMPLIED	Атрибут не является обязательным
A	Атрибут не является обязательным; анализатор возвращает значение **A**, если атрибут не указан
#FIXED A	Атрибут не должен быть указан или должен быть равен **A**; но в любом случае анализатор возвращает значение **A**

Ниже представлены два типичных примера обозначения атрибутов.

```
<!ATTLIST font style (plain|bold|italic|bold-italic)"plain">
<!ATTLIST size unit CDATA #IMPLIED>
```

В первом примере для элемента разметки `font` указан атрибут `style`, который может иметь четыре допустимых значения. По умолчанию используется значение `plain`. Во втором примере для элемента разметки **`size`** указан атрибут **`unit`**, который может содержать любую последовательность символов.

НА ЗАМЕТКУ! Для описания данных рекомендуется применять элементы разметки, а не атрибуты. В соответствии с этой рекомендацией стиль шрифта должен содержаться в отдельном элементе разметки, например, в следующем виде: **`<font><style>plain</style>... </font>`**. Но атрибуты обладают несомненным преимуществом при использовании перечислений, потому что анализатор может проверять допустимость тех или иных переменных. Так, если стиль шрифта является атрибутом, то анализатор проверяет наличие найденного стиля среди четырех указанных допустимых значений и выбирает значение по умолчанию, если ничего не указано.

Обработка атрибута типа `CDATA` несколько отличается от обработки атрибута типа `#PCDATA` и практически никак не связана с разделами `<![CDATA[...]]>`. Значение атрибута сначала *нормализуется*, т.е. синтаксический анализатор обрабатывает ссылки на символы и сущности (как, например, `&#233;` или `<`) и заменяет разделители пробелами.

Тип атрибута NMTOKEN (т.е. лексема имени) аналогичен типу CDATA, но в нем не допускается использование большинства символов, отличающихся от букв и цифр, а также разделителей в виде внутренних пробелов. Синтаксический анализатор удаляет все начальные и конечные пробелы как разделители. Тип атрибута NMTOKENS представляет собой список лексем имен, разделяемых пробелами.

Тип атрибута ID означает лексему имени, однозначную для данного документа. Однозначность лексемы имени проверяется синтаксическим анализатором. Применение данного типа атрибута демонстрируется в рассматриваемом далее примере программы. Тип атрибута IDREF означает ссылку на идентификатор, уже существующий в данном документе, наличие которого также проверяется синтаксическим анализатором. Тип атрибута IDREFS обозначает список ссылок на идентификаторы.

Атрибут типа ENTITY указывает на "непроанализированную внешнюю сущность". Этот тип унаследован от SGML и редко применяется на практике. В спецификации языка XML, доступной по адресу www.xml.com/axml/axml.html, приводится пример применения подобного типа атрибута.

Определение DTD может также содержать определения *сущностей*, или сокращений, которые заменяются в процессе синтаксического анализа. Характерный пример применения сущностей можно найти в описании пользовательского интерфейса веб-браузера Firefox. Данное описание отформатировано в соответствии с требованиями XML и содержит определения сущностей, подобные приведенному ниже.

```
<!ENTITY back.label "Back">
```

Далее в тексте документа могут встретиться ссылки на эти сущности, как показано в следующем примере:

```
<menuitem label="&back.label;"/>
```

В таком случае синтаксический анализатор заменяет ссылку на сущность замещающей строкой. Для интернационализации прикладной программы потребуется лишь изменить определение самой сущности. Другие примеры применения сущностей более сложны и менее распространены. Дополнительные сведения по данному вопросу можно найти в спецификации языка XML по указанному выше адресу.

На этом краткое введение в определения DTD завершается. Рассмотрим далее способы настройки синтаксического анализатора, чтобы воспользоваться всеми преимуществами DTD. Для этого необходимо сначала сообщить фабрике построителей документов о необходимости активизировать проверку следующим образом:

```
factory.setValidating(true);
```

Все построители документов, производимые этой фабрикой, проверяют соответствие входных данных определению DTD. Наиболее полезным результатом такой проверки является игнорирование всех разделителей в элементе разметки. Рассмотрим в качестве примера следующий фрагмент XML-разметки:

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

Синтаксический анализатор, не выполняющий проверку, возвращает все разделители между элементами разметки font, name и size. Он поступает так

потому, что ему неизвестно, какое из следующих правил описывает дочерние узлы элемента разметки `font`:

```
(name,size)
(#PCDATA,name,size)*
```

или

```
ANY
```

Если же в определении DTD указано правило (**`name, size`**), то синтаксическому анализатору должно быть известно, что разделитель этих элементов не относится к тексту. Построитель документа не будет учитывать разделители в узлах текста, если вызвать следующий метод:

```
factory.setIgnoringElementContentWhitespace(true);
```

Теперь нет никаких сомнений, что узел **`font`** имеет два дочерних узла. Следовательно, нет никаких оснований организовывать приведенный ниже громоздкий цикл.

```
for (int i = 0; i < children.getLength(); i++)
{
  Node child = children.item(i);
  if (child instanceof Element)
  {
    var childElement = (Element) child;
    if (childElement.getTagName().equals("name")) . . .
    else if (childElement.getTagName().equals("size")) . . .
  }
}
```

Вместо этого для доступа к первому и второму дочерним узлам можно написать следующий фрагмент кода:

```
Element nameElement = (Element) children.item(0);
Element sizeElement = (Element) children.item(1);
```

Именно в этих случаях особенно полезны определения DTD. Как видите, используя определения DTD, можно не включать больше в программу сложные фрагменты кода, выполняющие проверку, потому что синтаксический анализатор проделает эту работу при получении документа.

Если синтаксический анализатор сообщит об ошибке, прикладная программа должна каким-то образом отреагировать на это: зарегистрировать ошибку, сообщить о ней пользователю или сгенерировать исключение, чтобы прекратить синтаксический анализ. Поэтому, пользуясь всякий раз средствами проверки содержимого XML-документа, следует также установить обработчик исключений, создав объект, класс которого реализует интерфейс `ErrorHandler`. В этом интерфейсе объявлены следующие методы:

```
void warning(SAXParseException exception)
void error(SAXParseException exception)
void fatalError(SAXParseException exception)
```

Для установки обработчика исключений служит следующий метод `setErrorHandler()` из класса `DocumentBuilder`:

```
builder.setErrorHandler(handler);
```

javax.xml.parsers.DocumentBuilder 1.4

- **void setEntityResolver(EntityResolver *resolver*)**
  Устанавливает определитель сущностей, упоминаемых в анализируемых XML-документах.
- **void setErrorHandler(ErrorHandler *handler*)**
  Устанавливает обработчик исключений для выдачи предупреждений и сообщений об ошибках, возникающих при синтаксическом анализе XML-документов.

org.xml.sax.EntityResolver 1.4

- **public InputSource resolveEntity(String *publicID*, String *systemID*)**
  Возвращает источник вводимых данных, содержащий данные, определяемые заданными идентификаторами, или пустое значение **null**, указывающее на то, что определителю сущностей неизвестно, как обработать данное конкретное имя. Параметр ***publicID*** может принимать пустое значение **null**, если открытые идентификаторы не предоставляются.

org.xml.sax.InputSource 1.4

- **InputSource(InputStream *in*)**
- **InputSource(Reader *in*)**
- **InputSource(String *systemID*)**
  Создают источник вводимых данных на основании указанного потока ввода, потока чтения или системного идентификатора (обычно это относительный или абсолютный URL).

org.xml.sax.ErrorHandler 1.4

- **void fatalError(SAXParseException *exception*)**
- **void error(SAXParseException *exception*)**
- **void warning(SAXParseException *exception*)**
  Эти методы следует переопределить для создания собственного обработчика неустранимых ошибок, устранимых ошибок или предупреждений.

org.xml.sax.SAXParseException 1.4

- **int getLineNumber()**
- **int getColumnNumber()**
  Возвращают номер строки или столбца в конце вводимых данных, при обработке которых возникло исключение.

javax.xml.catalog.CatalogManager 9

- **static CatalogResolver catalogResolver(CatalogFeatures *features*, URI... *uris*)**

  Создает определитель сущностей, использующий файлы каталогов, доступные по указанным URL. Этот класс реализует интерфейс **EntityResolver**, а также классы определителей сущностей, применяемые в схеме StAX проверки XML-документов и XSLT-преобразованиях.

javax.xml.catalog.CatalogFeatures 9

- **static CatalogFeatures defaults()**

  Возвращает экземпляр с установками по умолчанию.

javax.xml.parsers.DocumentBuilderFactory 1.4

- **boolean isValidating()**
- **void setValidating(boolean *value*)**

  Возвращают или устанавливают свойство **validating** для фабрики. Если это свойство принимает логическое значение **true**, то созданные фабрикой синтаксические анализаторы будут выполнять проверку входных данных.

- **boolean isIgnoringElementContentWhitespace()**
- **void setIgnoringElementContentWhitespace(boolean *value*)**

  Получают или устанавливают значение, определяющее, следует ли игнорировать разделители между элементами разметки. Логическое значение **true** указывает на то, что созданные фабрикой синтаксические анализаторы будут игнорировать разделители в том случае, если для элемента разметки не задано смешанное содержимое (т.е. сочетание элементов разметки с атрибутами типа **#PCDATA**).

3.4.2. Схема XML-документов

Синтаксис языка XML Schema сложнее, чем у определений DTD, поэтому рассмотрим лишь самые основные его элементы. Дополнительные сведения о нем можно найти в учебном пособии, доступном по адресу `http://www.w3.org/TR/xmlschema-0`. Чтобы включить в документ ссылку на файл схемы типа XML Schema, в корневом элементе следует указать соответствующие атрибуты, как демонстрируется в приведенном ниже примере.

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="config.xsd">
  . . .
</config>
```

В данном примере объявления указывается, что при проверке документа должен использоваться файл схемы `config.xsd`. Если же в XML-документе

применяются пространства имен, то синтаксис немного усложняется. Подробнее об этом можно узнать из учебного пособия по указанному выше адресу. (Префикс `xsi` обозначает *псевдоним пространства имен*, как поясняется далее, в разделе 3.6.)

Схема определяет тип каждого элемента и атрибута. *Простой тип* может быть представлен символьной строкой с дополнительными ограничениями, накладываемыми на ее содержимое, а все остальное относится к *сложному типу*. У элемента разметки простого типа могут вообще отсутствовать атрибуты и дочерние элементы, а иначе это элемент разметки сложного типа. Но в то же время атрибуты всегда имеют простой тип. Примеры простых типов, встроенных в XML Schema, приведены ниже.

```
xsd:string
xsd:int
xsd:boolean
```

НА ЗАМЕТКУ! Здесь и далее используется префикс **`xsd:`**, обозначающий пространство имен XML Schema Definition. Некоторые авторы применяют для этой же цели префикс **`xs:`**.

По желанию можно определить собственные простые типы. Ниже приведен пример определения перечислимого типа.

```
<xsd:simpleType name="StyleType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="PLAIN" />
    <xsd:enumeration value="BOLD" />
    <xsd:enumeration value="ITALIC" />
    <xsd:enumeration value="BOLD_ITALIC" />
  </xsd:restriction>
</xsd:simpleType>
```

Определяя элемент разметки, следует указать его тип следующим образом:

```
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="size" type="xsd:int"/>
<xsd:element name="style" type="StyleType"/>
```

Тип ограничивает возможные варианты содержимого элемента. Например, проверка следующих элементов разметки даст положительный результат:

```
<size>10</size>
<style>PLAIN</style>
```

Приведенные ниже элементы разметки будут отвергнуты синтаксическим анализатором.

```
<size>default</size>
<style>SLANTED</style>
```

Простые типы можно объединять в сложные:

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element ref="name"/>
    <xsd:element ref="size"/>
    <xsd:element ref="style"/>
```

```
  </xsd:sequence>
</xsd:complexType>
```

где `FontType` — последовательность элементов разметки `name`, `size` и `style`. В данном определении использован атрибут `ref`, для которого ссылки на определения находятся в схеме. Допускаются также вложенные определения, как в приведенном ниже примере.

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
    <xsd:element name="style" type="StyleType">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="PLAIN" />
          <xsd:enumeration value="BOLD" />
          <xsd:enumeration value="ITALIC" />
          <xsd:enumeration value="BOLD_ITALIC" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Обратите внимание на то, что для элемента `style` использовано *анонимное определение типа*. Языковая конструкция `xsd:sequence` является аналогом операции сцепления определений DTD, а конструкция `xsd:choice` равнозначна логической операции `|`. Так, приведенная ниже схема разметки заменяет тип `email | phone` в определении DTD.

```
<xsd:complexType name="contactinfo">
  <xsd:choice>
    <xsd:element ref="email"/>
    <xsd:element ref="phone"/>
  </xsd:choice>
</xsd:complexType>
```

Для повторяющихся элементов можно использовать атрибуты `minoccurs` и `maxoccurs`. Например, аналогом типа `item*` в определении DTD является следующая схема разметки:

```
<xsd:element name="item" type=". . ." minoccurs="0"
     maxoccurs="unbounded">
```

Для определения атрибутов в разметку определений `complexType` следует ввести элементы `xsd:attribute`, как показано ниже.

```
<xsd:element name="size">
  <xsd:complexType>
    . . .
    <xsd:attribute name="unit" type="xsd:string"
         use="optional" default="cm"/>
  </xsd:complexType>
</xsd:element>
```

Приведенной выше схеме разметки равнозначен следующий оператор в определении DTD:

```
<!ATTLIST size unit CDATA #IMPLIED "cm">
```

Определения элемента и типа схемы разметки размещаются в элементе `xsd:schema` следующим образом:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  . . .
</xsd:schema>
```

Синтаксический анализ XML-документа, разметка которого определяется по заданной схеме, выполняется практически так же, как и синтаксический анализ XML-документа, для разметки которого служит определение DTD, за исключением нескольких перечисленных ниже отличий.

1. Следует активизировать средства поддержки пространств имен в XML-файлах, даже если они не используются в XML-документах.

   ```
   factory.setNamespaceAware(true);
   ```

2. Необходимо подготовить фабрику для обработки схем разметки. Сделать это позволяют приведенные ниже выражения.

   ```
   final String JAXP_SCHEMA_LANGUAGE =
     "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
   final String W3C_XML_SCHEMA =
     "http://www.w3.org/2001/XMLSchema";
   factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
   ```

3.4.3. Практический пример применения XML-документов

В этом разделе рассматривается пример применения XML-документа на практике. Допустим, для конфигурирования прикладной программы требуются данные, в которых задаются произвольные объекты, а не только текстовые строки. Для получения экземпляра объекта предоставляются два механизма: конструктор и фабричный метод. В частности, ниже показано, как создать объект типа `Color`, используя конструктор.

```
<construct class="java.awt.Color">
  <int>55</int>
  <int>200</int>
  <int>100</int>
</construct>
```

Ниже показано, как добиться той же самой цели с помощью фабричного метода. Если имя фабричного метода опускается, то по умолчанию выбирается метод `getInstance()`.

```
<factory class="java.util.logging.Logger"
         method="getLogger">
  <string>com.horstmann.corejava</string>
</factory>
```

Как видите, в данном примере разметки имеются элементы, описывающие символьные строки и целые числа. Аналогичным образом можно организовать

поддержку логического типа boolean и других примитивных типов данных. Ради большей наглядности данного примера ниже демонстрируется еще один механизм для поддержки примитивных типов.

```
<value type="int">30</value>
```

Таким образом, конфигурация прикладной программы состоит из последовательности записей, причем у каждой имеется свой идентификатор и объект. Однозначность идентификаторов проверяется синтаксическим анализатором.

```
<config>
  <entry id="background">
    <construct class="java.awt.Color">
      <value type="int">55</value>
      <value type="int">200</value>
      <value type="int">100</value>
    </construct>
  </entry>
  . . .
</config>
```

Определение DTD, приведенное в листинге 3.4, оказывается очень простым, а равнозначная схема XML-документа представлена в листинге 3.5. В этой схеме можно организовать дополнительную проверку на наличие только целого значения в элементе типа int или логического значения в элементе типа boolean. Обратите внимание на применение в этой схеме конструкции xsd:group для определения тех частей сложных типов, которые используются повторно.

В программе из листинга 3.2 демонстрируется порядок синтаксического анализа файла конфигурации, а в листинге 3.3 приведен пример такого файла. В данной программе применяется схема XML-документа вместо его определения DTD, если выбрать файл, содержащий символьную строку "-schema".

Данный пример наглядно демонстрирует типичное применение формата XML, который оказывается достаточно надежным для выражения отношений. Синтаксический анализатор XML-разметки придает формату XML еще большую ценность, беря на себя все хлопоты по проверке достоверности и снабжению стандартными значениями настроек по умолчанию.

Листинг 3.2. Исходный код из файла **read/XMLReadTest.java**

```
1   package read;
2
3   import java.io.*;
4   import java.lang.reflect.*;
5   import java.util.*;
6
7   import javax.xml.parsers.*;
8
9   import org.w3c.dom.*;
10  import org.xml.sax.*;
11
12  /**
13   * В этой программе демонстрирует применение
14   * XML-файла для описания объектов Java
15   * @version 1.0 2018-04-03
```

```
16   * @author Cay Horstmann
17   */
18  public class XMLReadTest
19  {
20    public static void main(String[] args)
21       throws ParserConfigurationException, SAXException,
22             IOException, ReflectiveOperationException
23    {
24       String filename;
25       if (args.length == 0)
26       {
27         try (var in = new Scanner(System.in))
28         {
29           System.out.print("Input file: ");
30           filename = in.nextLine();
31         }
32       }
33       else
34        filename = args[0];
35
36      DocumentBuilderFactory factory =
37             DocumentBuilderFactory.newInstance();
38      factory.setValidating(true);
39
40      if (filename.contains("-schema"))
41      {
42        factory.setNamespaceAware(true);
43        final String JAXP_SCHEMA_LANGUAGE =
44              "http://java.sun.com/xml/jaxp/"
45            + "properties/schemaLanguage";
46        final String W3C_XML_SCHEMA =
47              "http://www.w3.org/2001/XMLSchema";
48        factory.setAttribute(JAXP_SCHEMA_LANGUAGE,
49                             W3C_XML_SCHEMA);
50      }
51
52      factory.setIgnoringElementContentWhitespace(true);
53
54      DocumentBuilder builder =
55             factory.newDocumentBuilder();
56
57      builder.setErrorHandler(new ErrorHandler()
58      {
59        public void warning(SAXParseException e)
60               throws SAXException
61        {
62          System.err.println("Warning: " + e.getMessage());
63        }
64
65        public void error(SAXParseException e)
66               throws SAXException
67        {
68          System.err.println("Error: " + e.getMessage());
69          System.exit(0);
70        }
71
72        public void fatalError(SAXParseException e)
```

```
73              throws SAXException
74        {
75          System.err.println("Fatal error: "
76                             + e.getMessage());
77          System.exit(0);
78        }
79      });
80
81      Document doc = builder.parse(filename);
82      Map<String, Object> config =
83          parseConfig(doc.getDocumentElement());
84      System.out.println(config);
85    }
87
88    private static Map<String,
89                  Object> parseConfig(Element e)
90            throws ReflectiveOperationException
91    {
92      var result = new HashMap<String, Object>();
93      NodeList children = e.getChildNodes();
94      for (int i = 0; i < children.getLength(); i++)
95      {
96        var child = (Element) children.item(i);
97        String name = child.getAttribute("id");
98        Object value = parseObject((Element)
99                       child.getFirstChild());
100       result.put(name, value);
101     }
102     return result;
103   }
104
105   private static Object parseObject(Element e)
106           throws ReflectiveOperationException
107   {
108     String tagName = e.getTagName();
109     if (tagName.equals("factory"))
110       return parseFactory(e);
111     else if (tagName.equals("construct"))
112       return parseConstruct(e);
113     else
114     {
115       String childData = ((CharacterData)
116             e.getFirstChild()).getData();
117       if (tagName.equals("int"))
118         return Integer.valueOf(childData);
119       else if (tagName.equals("boolean"))
120         return Boolean.valueOf(childData);
121       else
122         return childData;
123     }
124   }
125
126   private static Object parseFactory(Element e)
127           throws ReflectiveOperationException
128   {
129     String className = e.getAttribute("class");
130     String methodName = e.getAttribute("method");
```

```
131     Object[] args = parseArgs(e.getChildNodes());
132     Class<?>[] parameterTypes =
133               getParameterTypes(args);
134     Method method = Class.forName(className)
135           .getMethod(methodName, parameterTypes);
136     return method.invoke(null, args);
137   }
138
139   private static Object parseConstruct(Element e)
140           throws ReflectiveOperationException
141   {
142     String className = e.getAttribute("class");
143     Object[] args = parseArgs(e.getChildNodes());
144     Class<?>[] parameterTypes =
145               getParameterTypes(args);
146     Constructor<?> constructor =
147           Class.forName(className)
148                 .getConstructor(parameterTypes);
149     return constructor.newInstance(args);
150   }
151
152   private static Object[] parseArgs(NodeList elements)
153           throws ReflectiveOperationException
154   {
155     var result = new Object[elements.getLength()];
156     for (int i = 0; i < result.length; i++)
157       result[i] = parseObject((Element)
158                   elements.item(i));
159       return result;
160   }
161
162   private static Map<Class<?>, Class<?>> toPrimitive =
163                 Map.of(Integer.class, int.class,
164                        Boolean.class, boolean.class);
165
166   private static Class<?>[]
167           getParameterTypes(Object[] args)
168   {
169     var result = new Class<?>[args.length];
170     for (int i = 0; i < result.length; i++)
171     {
172       Class<?> cl = args[i].getClass();
173       result[i] = toPrimitive.get(cl);
174       if (result[i] == null) result[i] = cl;
175     }
176     return result;
177   }
178 }
```

Листинг 3.3. Исходный код из файла **read/config.xml**

```
1  <?xml version="1.0"?>
2  <!DOCTYPE config SYSTEM "config.dtd">
3  <config>
4    <entry id="background">
5      <construct class="java.awt.Color">
```

```
6         <int>55</int>
7         <int>200</int>
8         <int>100</int>
9       </construct>
10    </entry>
11    <entry id="currency">
12      <factory class="java.util.Currency">
13        <string>USD</string>
14      </factory>
15    </entry>
16  </config>
```

Листинг 3.4. Исходный код из файла **read/config.dtd**

```
1  <!ELEMENT config (entry)*>
2
3  <!ELEMENT entry (string|int|boolean|construct|factory)>
4  <!ATTLIST entry id ID #IMPLIED>
5
6  <!ELEMENT construct
7            (string|int|boolean|construct|factory)*>
8  <!ATTLIST construct class CDATA #IMPLIED>
9
10 <!ELEMENT factory
11           (string|int|boolean|construct|factory)*>
12 <!ATTLIST factory class CDATA #IMPLIED>
13 <!ATTLIST factory method CDATA "getInstance">
14
15 <!ELEMENT string (#PCDATA)>
16 <!ELEMENT int (#PCDATA)>
17 <!ELEMENT boolean (#PCDATA)>
```

Листинг 3.5. Исходный код из файла **read/config.xsd**

```
1  <xsd:schema xmlns:xsd=
2              "http://www.w3.org/2001/XMLSchema">
3    <xsd:element name="config">
4      <xsd:complexType>
5        <xsd:sequence>
6          <xsd:element name="entry" minOccurs="0"
7                  maxOccurs="unbounded">
8           <xsd:complexType>
9             <xsd:group ref="Object"/>
10            <xsd:attribute name="id" type="xsd:ID"/>
11          </xsd:complexType>
12        </xsd:element>
13      </xsd:sequence>
14    </xsd:complexType>
15    </xsd:element>
16
17    <xsd:element name="construct">
18      <xsd:complexType>
19        <xsd:group ref="Arguments"/>
20        <xsd:attribute name="class"
```

```
                    type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="factory">
    <xsd:complexType>
      <xsd:group ref="Arguments"/>
      <xsd:attribute name="class"
                    type="xsd:string"/>
      <xsd:attribute name="method"
                    type="xsd:string"
                 default="getInstance"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:group name="Object">
    <xsd:choice>
     <xsd:element ref="construct"/>
      <xsd:element ref="factory"/>
      <xsd:element name="string" type="xsd:string"/>
      <xsd:element name="int" type="xsd:int"/>
      <xsd:element name="boolean" type="xsd:boolean"/>
    </xsd:choice>
  </xsd:group>

  <xsd:group name="Arguments">
    <xsd:sequence>
      <xsd:group ref="Object" minOccurs="0"
           maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:group>
 </xsd:schema>
```

3.5. Поиск информации средствами XPath

Если требуется найти информацию в XML-документе, придется организовать обход дерева DOM. Язык XPath упрощает доступ к узлам дерева. Допустим, имеется следующий XML-документ:

```
<html>
  <head>
    . . .
    <title>. . .</title>
    . . .
  </database>
  . . .
</html>
```

Чтобы получить из него имя пользователя базы данных, достаточно вычислить следующее выражение XPath:

```
/html/head/title/text()
```

Сделать это намного проще, чем организовывать непосредственный обход дерева DOM, выполнив перечисленные ниже действия.

1. Получить узел документа.
2. Получить первый дочерний элемент и привести его к типу `Element`.
3. Обнаружить элемент разметки `title` среди дочерних элементов.
4. Получить первый его дочерний элемент и привести его к типу узла `CharacterData`.
5. Получить из него данные.

Язык XPath позволяет описывать *ряд узлов* в XML-документе. Например, в следующем выражении описывается ряд элементов `form`, которые являются дочерними для элемента разметки `body`:

```
/html/body/form
```

Для выбора конкретного элемента служит операция `[]`. Так, в следующем выражении определяется первый дочерний элемент разметки (отчет индексов начинается с единицы):

```
/html/body/form[1]
```

Для получения значений атрибутов служит операция `@`. Например, в следующем выражении XPath описывается атрибут `action` первой таблицы:

```
/html/body/form[1]/@action
```

Наконец, в приведенном ниже выражении XPath описываются все узлы с атрибутами `action` всех элементов разметки `form`, которые являются дочерними для элемента разметки `body`.

```
/html/body/form/@action
```

В языке XPath имеется ряд функций, упрощающих работу с документом. Например, в следующем выражении определяется количество элементов `form`, дочерних для элемента разметки `body`:

```
count(/html/body/form)
```

Примеры выражений XPath, в том числе и довольно сложных, можно найти в спецификации этого языка, доступной по адресу `http://www.w3c.org/TR/xpath`. Имеется также очень удачно составленное руководство по XPath, доступное по адресу http://www.zvon.org/xxl/XPathTutorial/General_rus/examples.html.

Чтобы вычислить выражение XPath, необходимо создать сначала объект типа `XPath` средствами класса `XPathFactory`, как показано ниже.

```
XPathFactory xpfactory = XPathFactory.newInstance();
path = xpfactory.newXPath();
```

Затем вызывается приведенный ниже метод `evaluate()` для вычисления выражения XPath. Используя один объект типа `XPath`, можно обработать несколько выражений.

```
String username = path.evaluate(
      "/html/head/title/text()", doc);
```

В данной форме метод `evaluate()` возвращает результат в виде символьной строки. Это удобно для получения текста, например, из узла `title`

в приведенном выше примере. Если из выражения XPath получается ряд узлов, то для их обработки можно сделать следующий вызов:

```
XPathNodes result = path.evaluateExpression(
         "/html/body/form", doc, XPathNodes.class);
```

Класс `XPathNodes` подобен классу `NodeList`, но он расширяет интерфейс `Iterable`, давая возможность организовать расширенный цикл `for`. Такая возможность была внедрена в версии Java 9, а в прежних версиях пришлось бы сделать следующий вызов:

```
NodeList nodes = (NodeList) path.evaluate(
         "/html/body/form", doc, XPathConstants.NODESET);
```

Если же в итоге получается один узел, в таком случае можно сделать один из следующих вызовов:

```
Node node = path.evaluateExpression(
          "/html/body/form[1]", doc, Node.class);
node = (Node) path.evaluate("/html/body/form[1]", doc,
                            XPathConstants.NODE);
```

Если в итоге получается количество узлов, тогда можно воспользоваться следующим фрагментом кода:

```
int count = path.evaluateExpression(
          "count(/html/body/form)", doc, Integer.class);
count = ((Number) path.evaluate("count(/html/body/form)",
                 doc, XPathConstants.NUMBER)).intValue();
```

Поиск совсем не обязательно начинать с корневого узла документа. В качестве исходной точки можно выбрать любой узел и даже перечень узлов. Например, получив узел в результате вычисления приведенного выше выражения, можно сделать следующий вызов:

```
result = path.evaluate(expression, node);
```

В примере программы, исходный код которой приведен в листинге 3.6, демонстрируется порядок вычисления выражений XPath. Загрузите сначала XML-файл и введите выражение. Результат вычисления введенного выражения появится в нижней части окна.

Листинг 3.6. Исходный код из файла **xpath/XPathTester.java**

```
1  package xpath;
2
3  import java.io.*;
4  import java.nio.file.*;
5  import java.util.*;
6
7  import javax.xml.catalog.*;
8  import javax.xml.parsers.*;
9  import javax.xml.xpath.*;
10
11 import org.w3c.dom.*;
12 import org.xml.sax.*;
13
```

```
/**
 * В этой программе вычисляются выражения XPath
 * @version 1.1 2018-04-06
 * @author Cay Horstmann
 */
public class XPathTest
{
  public static void main(String[] args)
        throws Exception
  {
    DocumentBuilderFactory factory =
          DocumentBuilderFactory.newInstance();
    DocumentBuilder builder =
          factory.newDocumentBuilder();

    // избежать задержек при синтаксическом
    // анализе XHTML-файла; см. первое
    // примечание, приведенное в разделе 3.3.1
    builder.setEntityResolver(
          CatalogManager.catalogResolver(
                CatalogFeatures.defaults(),
          Paths.get("xpath/catalog.xml")
               .toAbsolutePath().toUri()));

    XPathFactory xpfactory = XPathFactory.newInstance();
    XPath path = xpfactory.newXPath();
    try (var in = new Scanner(System.in))
    {
      String filename;
      if (args.length == 0)
      {
        System.out.print("Input file: ");
        filename = in.nextLine();
      }
      else
        filename = args[0];

      Document doc = builder.parse(filename);
      var done = false;
      while (!done)
      {
        System.out.print(
            "XPath expression (empty line to exit): " );
        String expression = in.nextLine();
        if (expression.trim().isEmpty())
          done = true;
        else
        {
          try
          {
            XPathEvaluationResult<?> result =
               path.evaluateExpression(expression, doc);
            if (result.type() == XPathEvaluationResult
                                 .XPathResultType.NODESET)
```

```
68              {
69                for (Node n : (XPathNodes) result.value())
70                  System.out.println(description(n));
71              }
72              else if (result.type() ==
73                       XPathEvaluationResult
74                      .XPathResultType.NODESET)
75                System.out.println((Node) result.value());
76              else
77                System.out.println(result.value());
78            }
79            catch (XPathExpressionException e)
80            {
81              System.out.println(e.getMessage());
82            }
83          }
84        }
85      }
86    }
87
88    public static String description(Node n)
89    {
90      if (n instanceof Element)
91        return "Element " + n.getNodeName();
92      else if (n instanceof Attr)
93        return "Attribute " + n;
94      else
95        return n.toString();
96    }
97  }
```

javax.xml.xpath.XPathFactory 5.0

- **static XPathFactory newInstance()**

  Возвращает экземпляр класса **XPathFactory**, используемый для создания объектов типа **XPath**.

- **XPath newXpath()**

  Создает объект типа **XPath**, который можно использовать для обработки выражений XPath.

javax.xml.xpath.XPath 5.0

- **String evaluate(String *expression*, Object *startingPoint*)**

  Вычисляет выражение, начиная поиск с заданной исходной точки. В качестве исходной точки может быть указан узел или перечень узлов. Если в результате вычисления данного выражения получается узел или ряд узлов, то возвращаемая символьная строка содержит данные из всех дочерних текстовых узлов.

javax.xml.xpath.XPath 5.0 (окончание)

- **Object evaluate(String *expression*, Object *startingPoint*, QName *resultType*)**

  Вычисляет выражение, начиная поиск с заданной исходной точки. В качестве исходной точки может быть указан узел или перечень узлов. В качестве параметра ***resultType*** задается одна из следующих констант, определяемых в классе **XPathConstants**: **STRING**, **NODE**, **NODESET**, **NUMBER** или **BOOLEAN**. Возвращаемое значение относится к типу **String**, **Node**, **NodeList**, **Number** или **Boolean**.

- **<T> T evaluateExpression(String *expression*, Object *item*, Class<T> type)** 9

  Вычисляет заданное выражение и возвращает результат в виде значения указанного типа.

- **XPathEvaluationResult<?> evaluateExpression(String *expression*, InputSource *source*)** 9

  Вычисляет заданное выражение.

javax.xml.xpath.XPathEvaluationResult<T> 9

- **XPathEvaluationResult.XPathResultType type()**

  Возвращает одну из следующих перечислимых констант: **STRING**, **NODESET**, **NODE**, **NUMBER**, **BOOLEAN**.

- **T value()**

  Возвращает результирующее значение.

3.6. Использование пространств имен

Во избежание конфликтов при использовании одинаковых имен в языке Java предусмотрены пакеты. Разные классы могут иметь одинаковые имена, если они находятся в разных пакетах. В XML для различения одинаково именуемых элементов и атрибутов используется механизм *пространств имен*. Пространство имен обозначается с помощью универсального идентификатора ресурсов (URI), как демонстрируется в приведенном ниже примере.

```
http://www.w3.org/2001/XMLSchema
uuid:1c759aed-b748-475c-ab68-10679700c4f2
urn:com:books-r-us
```

Чаще всего для этой цели используется формат URL по сетевому протоколу HTTP. Следует, однако, иметь в виду, что URL в данном случае выполняет лишь роль идентификатора. Приведенные ниже URL обозначают *разные* пространства имен, хотя веб-сервер интерпретировал бы их как указатели на один и тот же документ.

```
http://www.horstmann.com/corejava
http://www.horstmann.com/corejava/index.html
```

Более того, URL, определяющий пространство имен, может не указывать на конкретный документ. XML-анализатор и не пытается найти что-нибудь по этому адресу. Тем не менее по URL, обозначающему пространство имен, принято располагать документ с описанием назначения этого пространства. Например, по адресу `http://www.w3c.org/2001/XMLSchema` находится документ с описанием стандарта XML Schema.

А зачем для обозначения пространства имен применяются URL? Очевидно, что в таком случае легче гарантировать их однозначность. В самом деле, для подлинного URL однозначность имени узла сети гарантируется структурой системы доменных имен, а однозначность остальной части URL должна обеспечить надлежащая организация файловой системы. Именно из этих соображений для многих пакетов выбраны доменные имена с обратным порядком следования доменов.

Безусловно, обеспечить однозначность для длинных имен проще, но использовать их в программе не совсем удобно. В языке Java для этой цели предусмотрен механизм импорта пакетов, в результате чего в исходном тексте программы присутствуют в основном имена классов. Аналогичный механизм предусмотрен и в XML, как показано ниже. В итоге элемент и все его дочерние узлы становятся частью заданного пространства имен.

```
<элемент xmlns="URI_пространства_имен">
  дочерные_узлы
</элемент>
```

При необходимости дочерний узел может обеспечить себе отдельное пространство имен, как показано в приведенном ниже примере. В данном случае первый дочерний узел и внучатые узлы младшего уровня принадлежат второму пространству имен.

```
<элемент xmlns="URI_пространства_имен_1">
  <дочерний_узел xmlns="URI_пространства_имен_2">
    внучатые_узлы
  </дочерний_узел>
    другие дочерние_узлы
</элемент>
```

Этот простой механизм подходит только в том случае, если требуется одно пространство имен или же если пространства имен вложены друг в друга естественным образом. В иных случаях более предпочтительным оказывается альтернативный механизм, аналог которого отсутствует в Java. Для пространства имен можно использовать *префикс*, т.е. короткий идентификатор, выбираемый для конкретного документа. Ниже приведен типичный пример применения префикса `xsd` в файле схемы типа XML Schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="config"/>
  . . .
</xsd:schema>
```

Атрибут `xmlns:`*`префикс`*`="`*`URI_пространства_имен`*`"` определяет пространство имен и префикс. В данном примере префиксом является символьная строка `"xsd"`. Таким образом, атрибут `xsd:schema` фактически означает следующее: указанная схема (`schema`) находится в пространстве имен `http://www.w3.org/2001/XMLSchema`.

НА ЗАМЕТКУ! Пространство имен родительского элемента наследуется только дочерними элементами. Атрибуты без явного указания префикса не считаются частью пространства имен. Рассмотрим следующий вымышленный пример:

```
<config xmlns="http://www.horstmann.com/corejava"
  xmlns:si="http://www.bipm.fr/enus/3_SI/si.html">
  <size value="210" si:unit="mm"/>
  . . .
</config>
```

В данном примере элементы разметки **config** и **size** являются частью пространства имен, определяемого по следующему URI: **http://www.horstmann.com/corejava**. Атрибут **si:unit** является частью пространства имен по следующему URI: **http://www.bipm.fr/enus/3_SI/si.html**. Но атрибут **value** *не* принадлежит ни одному из этих пространств имен.

Манипулирование пространствами имен в синтаксическом анализаторе поддается контролю. По умолчанию пространства имен не принимаются во внимание в DOM-анализаторе. Чтобы включить режим обработки пространств имен, достаточно вызвать метод `setNamespaceAware()` из класса `DocumentBuilderFactory` следующим образом:

```
factory.setNamespaceAware(true);
```

После этого все созданные данной фабрикой конструкторы будут поддерживать пространства имен. У каждого узла имеются следующие три свойства.

- *Уточненное имя* с префиксом, возвращаемое методами `getNodeName()`, `getTagName()` и т.д.
- URI пространства имен, возвращаемый методом `getNamespaceURI()`.
- *Локальное имя* без префикса, возвращаемое методом `getLocalName()`.

Обратимся к конкретному примеру. Допустим, синтаксический анализатор обнаруживает следующий элемент разметки:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

В таком случае он сообщает о наличии следующих свойств узла.

- Уточненное имя: `xsd:schema`.
- URI пространства имен: `http://www.w3.org/2001/XMLSchema`.
- Локальное имя: `schema`.

НА ЗАМЕТКУ! Если режим учета и обработки пространств имен отключен, методы **getNamespaceURI()** и **getLocalName()** возвращают пустое значение **null**.

org.w3c.dom.Node 1.4

- **String getLocalName()**

  Возвращает локальное имя (без префикса) или пустое значение **null**, если режим обработки пространств имен отключен.

org.w3c.dom.Node 1.4 *(окончание)*

- **String getNamespaceURI()**

  Возвращает URI пространства имен или пустое значение **null**, если узел не является частью пространства имен или режим обработки пространств имен отключен.

javax.xml.parsers.DocumentBuilderFactory 1.4

- **boolean isNamespaceAware()**
- **void setNamespaceAware(boolean *value*)**

  Получают или устанавливают значение свойства, определяющего режим обработки пространств имен. Если установлено логическое значение **true** параметра ***value***, то в генерируемых фабрикой синтаксических анализаторах будет включен режим обработки пространств имен.

3.7. Потоковые синтаксические анализаторы

DOM-анализатор считывает XML-документ и представляет его в виде древовидной структуры данных. Для большинства приложений оказывается достаточно и модели DOM. Но она неэффективна, если документ крупный, а алгоритм его обработки слишком прост, чтобы оперативно анализировать узлы, не просматривая все дерево в целом. В подобных случаях следует применять потоковые синтаксические анализаторы.

В последующих разделах будут рассмотрены потоковые анализаторы, доступные в библиотеке Java: "почтенный" SAX-анализатор и более современный StAX-анализатор, который появился в версии Java 6. SAX-анализатор использует обратные вызовы событий, а StAX-анализатор предоставляет итератор событий синтаксического анализа. Последний оказывается более удобным в употреблении.

3.7.1. Применение SAX-анализатора

SAX-анализатор уведомляет о событиях, наступающих в ходе синтаксического анализа компонентов данных, вводимых из XML-документа. Но сам документ не хранится в памяти, а создание структуры из вводимых данных возлагается на обработчики событий. На самом деле в основу работы DOM-анализатора положен тот же принцип, что и для SAX-анализатора: дерево модели DOM строится по мере приема событий, наступающих при синтаксическом анализе.

Чтобы воспользоваться SAX-анализатором, необходимо создать обработчик событий, определяющий действия для обработки различных событий, наступающих при синтаксическом анализе. В интерфейсе `ContentHandler` определен ряд перечисленных ниже методов обратного вызова, к которым обращается анализатор в ходе синтаксического анализа XML-документа.

- Методы `startElement()` и `endElement()` вызываются всякий раз, когда получается открывающий и закрывающий дескриптор.

- Метод `characters()` вызывается при получении символьных данных.
- Методы `startDocument()` и `endDocument()` вызываются в начале и в конце документа.

Например, при синтаксическом анализе следующего фрагмента разметки:

```
<font>
  <name>Helvetica</name>
  <size units="pt">36</size>
</font>
```

SAX-анализатор генерирует обратные вызовы перечисленных ниже методов.

1. Метод `startElement()`, имя элемента разметки: `font`.
2. Метод `startElement()`, имя элемента разметки: `name`.
3. Метод `characters()`, содержимое: `Helvetica`.
4. Метод `endElement()`, имя элемента разметки: `name`.
5. Метод `startElement()`, имя элемента разметки: `size`, атрибуты: `units="pt"`.
6. Метод `characters()`, содержимое: `36`.
7. Метод `endElement()`, имя элемента разметки: `size`.
8. Метод `endElement()`, имя элемента разметки: `font`.

Для выполнения требуемых действий при синтаксическом анализе содержимого вводимого файла необходимо переопределить эти методы. В примере программы, исходный код которой приведен в конце этого раздела, выводятся сведения обо всех гипертекстовых ссылках типа `<a href="...">`, найденных в HTML-файле. В ней переопределяется метод `startElement()` обработчика событий и реализуется проверка всех гипертекстовых ссылок с именем `a` и атрибутом `href`. Подобный код применяется в поисковых роботах, которые автоматически выявляют новые веб-страницы для индексации, переходя по ссылкам.

НА ЗАМЕТКУ! К сожалению, многие HTML-страницы совершенно не соответствуют формату XML, и поэтому рассматриваемая здесь программа не способна произвести их синтаксический анализ. Тем не менее большинство веб-страниц, созданных консорциумом W3C, составлены на XHTML (диалекте HTML, соответствующем формату XML). Поэтому этими веб-страницами можно воспользоваться для тестирования данной программы. Например, для составления списка всех гипертекстовых ссылок по соответствующим URL на веб-странице, доступной по адресу **`http://www.w3c.org/MarkUp`**, необходимо ввести следующую команду:

```
java SAXTest http://www.w3c.org/MarkUp
```

Рассматриваемая здесь программа служит характерным примером употребления SAX-анализатора. В ней полностью игнорируется контекст, в котором находится элемент разметки `a`, а также не сохраняется древовидная структура документа. Для получения SAX-анализатора используется приведенный ниже фрагмент кода.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
```

После этого документ можно обработать следующим образом:

```
parser.parse(source, handler)
```

где `source` — источник входных данных, который может быть файлом, символьной строкой с URL или потоком ввода, а `handler` относится к подклассу, производному от класса `DefaultHandler`. В классе `DefaultHandler` определены методы, объявленные в следующих интерфейсах:

```
ContentHandler
DTDHandler
EntityResolver
ErrorHandler
```

Эти методы не выполняют никаких действий. В рассматриваемой здесь программе определен обработчик событий, в котором для поиска элементов с именем `a` и атрибутом `href` переопределяется метод `startElement()` из интерфейса `ContentHahdler`:

```
var handler = new DefaultHandler()
  {
    public void startElement(String namespaceURI,
          String lname, String qname, Attributes attrs)
    {
      if (lname.equalsIgnoreCase("a") && attrs != null)
      {
        for (int i = 0; i < attrs.getLength(); i++)
        {
          String aname = attrs.getLocalName(i);
          if (aname.equalsIgnoreCase("href"))
            System.out.println(attrs.getValue(i));
        }
      }
    }
};
```

Методу `startElement()` передаются три параметра, описывающие имя элемента. В частности, параметр `qname` сообщает уточненное имя в форме `префикс:локальное_имя`. Если включен режим обработки пространств имен, то параметры `namespaceURI` и `lname` описывают пространство имен и локальное (неуточненное) имя.

Как и при использовании DOM-анализатора, режим обработки пространств имен исходно отключен. Для его включения достаточно вызвать метод `setNamespaceAware()` из фабричного класса `SAXParserFactory` следующим образом:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
SAXParser saxParser = factory.newSAXParser();
```

В рассматриваемой здесь программе преодолевается еще одно распространенное препятствие. В начале XHTML-файла обычно находится дескриптор, содержащий ссылку на описание DTD, которое требуется загрузить синтаксическому анализатору. Очевидно, что консорциуму W3C явно не улыбалась перспектива обслуживать миллиарды копий файлов вроде `www.w3.org/TR/xhtml1/DTD/`

xhtml1-strict.dtd. Поэтому они вообще отказались от такого обслуживания, но на момент написания этой книги описание DTD все же обслуживалось, хотя и очень медленно. Если же вам не требуется проверка достоверности документа, просто сделайте следующий вызов:

```
factory.setFeature("http://apache.org/xml/features/"
                + "nonvalidating/load-external-dtd", false);
```

Итак, в листинге 3.7 приведен исходный код простейшего поискового робота. Далее в этой главе рассматривается еще один интересный пример применения SAX-анализатора. Чтобы превратить источник данных, несовместимый с форматом XML, в XML-документ, проще всего уведомить о SAX-событиях, о которых будет затем сообщать сам XML-анализатор. Более подробно эти вопросы рассматриваются в разделе 3.9.

Листинг 3.7. Исходный код из файла **sax/SAXTest.java**

```
1  package sax;
2
3  import java.io.*;
4  import java.net.*;
5  import javax.xml.parsers.*;
6  import org.xml.sax.*;
7  import org.xml.sax.helpers.*;
8
9  /**
10  * В этой программе демонстрируется применение
11  * SAX-анализатора. Программа выводит все гиперссылки
12  * из веб-страницы формата XHTML.
13  * Использование: java SAXTest url
14  * @version 1.01 2018-05-01
15  * @author Cay Horstmann
16  */
17 public class SAXTest
18 {
19   public static void main(String[] args) throws Exception
20   {
21     String url;
22     if (args.length == 0)
23     {
24       url = "http://www.w3c.org";
25        System.out.println("Using " + url);
26     }
27     else url = args[0];
28
29     var handler = new DefaultHandler()
30     {
31       public void startElement(String namespaceURI,
32           String lname, String qname, Attributes attrs)
33       {
34         if (lname.equals("a") && attrs != null)
35         {
36           for (int i = 0; i < attrs.getLength(); i++)
37           {
```

```
38              String aname = attrs.getLocalName(i);
39              if (aname.equals("href"))
40                System.out.println(attrs.getValue(i));
41            }
42          }
43        }
44      };
45
46      SAXParserFactory factory =
47               SAXParserFactory.newInstance();
48      factory.setNamespaceAware(true);
49      factory.setFeature("http://apache.org/xml/features/"
50                      + "nonvalidating/load-external-dtd",
51                        false);
52      SAXParser saxParser = factory.newSAXParser();
53      InputStream in = new URL(url).openStream();
54      saxParser.parse(in, handler);
55    }
56 }
```

javax.xml.parsers.SAXParserFactory 1.4

- **static SAXParserFactory newInstance()**

  Возвращает экземпляр класса **SAXParserFactory**.

- **SAXParser newSAXParser()**

  Возвращает экземпляр класса **SAXParser**.

- **boolean isNamespaceAware()**
- **void setNamespaceAware(boolean *value*)**

  Получают или устанавливают значение свойства **namespaceAware**, определяющего режим учета и обработки пространств имен в фабрике. Если в этом свойстве установлено логическое значение **true**, то режим учета и обработки пространств имен активизирован для синтаксических анализаторов, генерируемых фабрикой.

- **boolean isValidating()**
- **void setValidating(boolean *value*)**

  Получают или устанавливают значение свойства **validating**, определяющего режим проверки достоверности вводимых данных в фабрике. Если в этом свойстве установлено логическое значение **true**, то режим проверки достоверности вводимых данных активизирован для синтаксических анализаторов, генерируемых фабрикой.

javax.xml.parsers.SAXParser 1.4

- **void parse(File *f*, DefaultHandler *handler*)**
- **void parse(String *url*, DefaultHandler *handler*)**
- **void parse(InputStream *in*, DefaultHandler *handler*)**

  Выполняют синтаксический анализ XML-документа, полученного из файла по указанному URL или из потока ввода, а также оповещают заданный обработчик о событиях, наступающих в ходе синтаксического анализа.

org.xml.sax.ContentHandler 1.4

- **void startDocument()**
- **void endDocument()**

  Вызываются в начале и в конце XML-документа соответственно.
- **void startElement(String *uri*, String *lname*, String *qname*, Attributes *attr*)**
- **void endElement(String *uri*, String *lname*, String *qname*)**

  Вызываются в начале и в конце элемента разметки соответственно. Если в анализаторе учитываются пространства имен, он сообщает URI пространства имен, локальное имя без префикса и полностью уточненное имя с префиксом.
- **void characters(char[] *data*, int *start*, int *length*)**

  Вызывается, когда анализатор сообщает символьные данные.

org.xml.sax.Attributes 1.4

- **int getLength()**

  Возвращает количество атрибутов, хранящихся в коллекции атрибутов.
- **String getLocalName(int *index*)**

  Возвращает локальное имя (без префикса) атрибута по указанному индексу или пустую символьную строку, если для синтаксического анализатора не включен режим обработки пространств имен.
- **String getURI(int *index*)**

  Возвращает URI пространства имен для атрибута по указанному индексу или пустую символьную строку, если узел не является частью пространства имен или же если для синтаксического анализатора не включен режим обработки пространств имен.
- **String getQName(int *index*)**

  Возвращает уточненное имя (с префиксом) атрибута по указанному индексу или пустую символьную строку, если уточненное имя не сообщено синтаксическим анализатором.
- **String getValue(int *index*)**
- **String getValue(String *qname*)**
- **String getValue(String *uri*, String *lname*)**

  Возвращают значение атрибута по указанному индексу, уточненное имя или URI пространства имен вместе с локальным именем. Если такое значение отсутствует, то возвращается пустое значение **null**.

3.7.2. Применение StAX-анализатора

StAX-анализатор является "извлекающим" синтаксическим анализатором. Вместо того чтобы устанавливать обработчик событий, достаточно произвести перебор событий в следующем цикле:

```
InputStream in = url.openStream();
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader parser = factory.createXMLStreamReader(in);
while (parser.hasNext())
```

```
{
  int event = parser.next();
  вызвать методы синтаксического анализатора parser,
  чтобы получить подробные сведения о событии
}
```

Например, в ходе синтаксического анализа следующего фрагмента разметки:

```
<font>
  <name>Helvetica</name>
  <size units="pt">36</size>
</font>
```

синтаксический анализатор выдает перечисленные ниже события.

1. START_ELEMENT, имя элемента: font.
2. CHARACTERS, содержимое: пробел.
3. START_ELEMENT, имя элемента разметки: name.
4. CHARACTERS, содержимое: Helvetica.
5. END_ELEMENT, имя элемента разметки: name.
6. CHARACTERS, содержимое: пробел.
7. START_ELEMENT, имя элемента разметки: size.
8. CHARACTERS, содержимое: 36.
9. END_ELEMENT, имя элемента разметки: size.
10. CHARACTERS, содержимое: пробел.
11. END_ELEMENT, имя элемента разметки: font.

Чтобы проанализировать значения атрибутов, следует вызвать соответствующие методы из класса XMLStreamReader. Например, при вызове следующего метода получается атрибут units текущего элемента:

```
String units = parser.getAttributeValue(null, "units");
```

По умолчанию режим обработки пространств имен включен. Отключить его можно, видоизменив фабрику следующим образом:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, false);
```

В листинге 3.8 приведен исходный код программы поискового робота, реализованной вместе со StAX-анализатором. Нетрудно заметить, что исходный код этой программы намного проще, чем код аналогичной программы со SAX-анализатором, поскольку в данном случае не требуется организовывать обработку событий.

Листинг 3.8. Исходный код из файла **stax/StAXTest.java**

```
1  package stax;
2
3  import java.io.*;
4  import java.net.*;
5  import javax.xml.stream.*;
```

```
6
7  /**
8   * В этой программе демонстрируется применение
9   * StAX-анализатора. Программа выводит все гиперссылки
10  * из веб-страницы формата XHTML.
11  * Использование: java StAXTest url
12  * @author Cay Horstmann
13  * @version 1.1 2018-05-01
14  */
15 public class StAXTest
16 {
17   public static void main(String[] args) throws Exception
18   {
19     String urlString;
20     if (args.length == 0)
21     {
22       urlString = "http://www.w3c.org";
23       System.out.println("Using " + urlString);
24     }
25     else urlString = args[0];
26     var url = new URL(urlString);
27     InputStream in = url.openStream();
28     XMLInputFactory factory =
29             XMLInputFactory.newInstance();
30     XMLStreamReader parser =
31                     factory.createXMLStreamReader(in);
32     while (parser.hasNext())
33     {
34       int event = parser.next();
35       if (event == XMLStreamConstants.START_ELEMENT)
36       {
37         if (parser.getLocalName().equals("a"))
38         {
39           String href =
40                   parser.getAttributeValue(null, "href");
41           if (href != null)
42             System.out.println(href);
43         }
44       }
42     }
43   }
44 }
```

javax.xml.stream.XMLInputFactory 6

- **static XMLInputFactory newInstance()**

  Возвращает экземпляр класса **XMLInputFactory**.

- **void setProperty(String *name*, Object *value*)**

  Задает свойство для данной фабрики или генерирует исключение типа **IllegalArgumentException**, если свойство не поддерживается или не допускает установку заданного значения. В реализации JDK поддерживаются следующие свойства, допускающие установку логических значений:

javax.xml.stream.XMLInputFactory 6 *(продолжение)*

"javax.xml.stream.isValidating"	При логическом значении **false** (по умолчанию) документ не проверяется. В спецификации это свойство не требуется
"javax.xml.stream.isNamespaceAware"	При логическом значении **true** (по умолчанию) обрабатываются пространства имен. В спецификации это свойство не требуется
"javax.xml.stream.isCoalescing"	При логическом значении **false** (по умолчанию) соседние символы не объединяются
"javax.xml.stream.isReplacingEntityReferences"	При логическом значении **true** (по умолчанию) ссылки на сущности заменяются и сообщаются в виде символьных данных
"javax.xml.stream.isSupportingExternalEntities"	При логическом значении **true** (по умолчанию) разрешаются внешние сущности. В спецификации не определяется никаких значений этого свойства по умолчанию
"javax.xml.stream.supportDTD"	При логическом значении **true** (по умолчанию) об описаниях DTD сообщается как о событиях

- **XMLStreamReader createXMLStreamReader(InputStream *in*)**
- **XMLStreamReader createXMLStreamReader(InputStream *in*, String *characterEncoding*)**
- **XMLStreamReader createXMLStreamReader(Reader *in*)**
- **XMLStreamReader createXMLStreamReader(Source *in*)**

Создают синтаксический анализатор, читающий данные из заданного потока ввода, потока чтения или источника JAXP.

javax.xml.stream.XMLStreamReader 6

- **boolean hasNext()**

  Возвращает логическое значение **true**, если существует другое событие синтаксического анализа.
- **int next()**

  Задает состояние синтаксического анализатора для последующего события синтаксического анализа и возвращает одну из следующих констант: **START_ELEMENT**, **END_ELEMENT**, **CHARACTERS**, **START_DOCUMENT**, **END_DOCUMENT**, **CDATA**, **COMMENT**, **SPACE** (игнорируемый пробел), **PROCESSING_INSTRUCTION**, **ENTITY_REFERENCE**, **DTD**.
- **boolean isStartElement()**
- **boolean isEndElement()**
- **boolean isCharacters()**
- **boolean isWhiteSpace()**

  Возвращают логическое значение **true**, если текущее событие связано с начальным элементом, конечным элементом, символьными данными или разделителем в виде пробела.
- **QName getName()**
- **String getLocalName()**

  Получают имя элемента в событии **START_ELEMENT** или **END_ELEMENT**.
- **String getText()**

  Возвращают символы события **CHARACTERS**, **COMMENT** или **CDATA**, замещающее значение для константы **ENTITY_REFERENCE** или внутреннее подмножество DTD.
- **int getAttributeCount()**
- **QName getAttributeName(int *index*)**
- **String getAttributeLocalName(int *index*)**
- **String getAttributeValue(int *index*)**

  Получают подсчет количества атрибутов, а также имена и значения атрибутов, при условии, что текущим оказывается событие **START_ELEMENT**.
- **String getAttributeValue(String *namespaceURI*, String *name*)**

  Получает значение атрибута по данному имени, при условии, что текущим оказывается событие **START_ELEMENT**. Если параметр ***namespaceURI*** принимает пустое значение **null**, пространство имен не проверяется.

3.8. Формирование XML-документов

Итак, рассмотрев способы написания программ на Java, предназначенных для чтения XML-документов, перейдем к способам написания программ, выполняющих обратное действие, т.е. формирующих данные для вывода в формате XML. Разумеется, XML-документ можно сформировать, введя в программу последовательность вызовов метода `print()` и выводя с их помощью элементы разметки, атрибуты и текст. Но для создания такого громоздкого кода потребуется много времени и труда. Кроме того, подобный код, как правило, изобилует ошибками. Очень легко, например, ошибиться при употреблении специальных знаков " или < в значениях атрибутов и в текстовом содержимом.

Намного удобнее создать дерево модели DOM, представляющей документ, а затем вывести его содержимое по месту назначения. Подробности такого подхода к формированию XML-документов обсуждаются в последующих разделах.

3.8.1. XML-документы без пространств имен

Чтобы построить древовидную структуру DOM, необходимо сформировать сначала пустой документ с помощью метода `newDocument()` из класса `DocumentBuilder` следующим образом:

```
Document doc = builder.newDocument();
```

Затем следует вызвать метод `createElement()` из класса `Document`, чтобы построить элементы документа, как показано ниже.

```
Element rootElement = doc.createElement(rootName);
Element childElement = doc.createElement(childName);
```

Далее создаются текстовые узлы с помощью метода `createTextNode()`:

```
Text textNode = doc.createTextNode(textContents);
```

3.8.2. XML-документы с пространствами имен

Если используются пространства имен, то процедура формирования XML-документа оказывается несколько иной. Сначала фабрика построителей документов устанавливается в режим управления пространствами имен, а затем создается построитель документов, как показано ниже.

```
DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
builder = factory.newDocumentBuilder();
```

Далее для создания любых узлов вместо метода `createElement()` вызывается метод `createElementNS()`:

```
String namespace = "http://www.w3.org/2000/svg";
Element rootElement = doc.createElementNS(namespace, "svg");
```

Если узел имеет уточненное имя с префиксом пространства имен, то любые требующиеся атрибуты с префиксом `xmlns` создаются автоматически. Так, если требуется ввести данные формата SVG в XHTML-документ, для этой цели можно построить соответствующий элемент аналогично приведенному ниже.

```
Element svgElement = doc.createElement(namespace, "svg:svg")
```

Когда этот элемент записывается, он превращается в следующий элемент разметки:

```
<svg:svg xmlns:svg="http://www.w3.org/2000/svg">
```

Если же требуется установить атрибуты элемента разметки, имена которых находятся в отдельном пространстве имен, вызывается метод `setAttributeNS()` из класса `Element`:

```
rootElement.setAttributeNS(namespace, qualifiedName, value);
```

3.8.3. Запись XML-документов

Как ни странно, записать дерево модели DOM в поток вывода не так-то просто. Для этой цели проще всего воспользоваться прикладным интерфейсом API языка XSLT (Extensible Stylesheet Language Transformations — расширяемый язык преобразования XML-документов). Более подробно язык XSLT рассматривается в последнем разделе этой главы, а до тех пор допустим, что приведенный ниже код каким-то волшебным образом позволяет получить данные, выводимые в формате XML.

Над документом выполняется холостое преобразование, а результат записывается в поток вывода. Чтобы включить узел `DOCTYPE` в выводимые данные, следует также указать идентификаторы `SYSTEM` и `PUBLIC` в качестве свойств вывода.

```
// построить объект холостого преобразования:
Transformer t = TransformerFactory.newInstance()
                                  .newTransformer();
// установить свойства вывода, чтобы получить узел DOCTYPE:
t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
                    systemIdentifier);
t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC,
                    publicIdentifier);
// установить отступ:
t.setOutputProperty(OutputKeys.INDENT, "yes");
t.setOutputProperty(OutputKeys.METHOD, "xml");
t.setOutputProperty(
     "{http://xml.apache.org/xslt}indent-amount", "2");
// выполнить холостое преобразование и вывести
// результат в файл:
t.transform(new DOMSource(doc),
            new StreamResult(new FileOutputStream(file)));
```

Еще один способ записи XML-документов состоит в применении интерфейса `LSSerializer`. Для получения экземпляра класса, реализующего этот интерфейс, служит следующий фрагмент кода:

```
DOMImplementation impl = doc.getImplementation();
DOMImplementationLS implLS = (DOMImplementationLS)
                             impl.getFeature("LS", "3.0");
LSSerializer ser = implLS.createLSSerializer();
```

Если требуется ввести пробелы и разрывы строк в документ, достаточно установить следующий флаг:

```
ser.getDomConfig().setParameter("format-pretty-print",
                                true);
```

И тогда преобразовать документ в символьную строку не составит особого труда:

```
String str = ser.writeToString(doc);
```

Если же требуется вывести документ непосредственно в файл, нужно создать объект типа `LSOutput` следующим образом:

```
LSOutput out = implLS.createLSOutput();
out.setEncoding("UTF-8");
```

```
out.setByteStream(Files.newOutputStream(path));
ser.write(doc, out);
```

javax.xml.parsers.DocumentBuilder 1.4

- **Document newDocument()**
  Возвращает пустой документ.

org.w3c.dom.Document 1.4

- **Element createElement(String *name*)**
- **Element createElementNS(String *uri*, String *qname*)**
  Создают элемент с заданным именем.
- **Text createTextNode(String *data*)**
  Создает текстовый узел с указанными данными.

org.w3c.dom.Node 1.4

- **Node appendChild(Node *child*)**
  Присоединяет узел к списку его дочерних узлов. Возвращает присоединенный узел.

org.w3c.dom.Element 1.4

- **void setAttribute(String *name*, String *value*)**
- **void setAttributeNS(String *uri*, String *qname*, String *value*)**
  Устанавливают заданное значение в атрибуте с указанным именем. Если в полностью уточненном имени имеется альтернативный префикс, то параметр ***uri*** должен принимать пустое значение **null**.

javax.xml.transform.TransformerFactory 1.4

- **static TransformerFactory newInstance()**
  Возвращает экземпляр класса **TransformerFactory**.
- **Transformer newTransformer()**
  Возвращает экземпляр класса **Transformer**, выполняющий тождественное (холостое) преобразование, не предполагающее никаких действий.

javax.xml.transform.Transformer 1.4

- **void setOutputProperty(String *name*, String *value*)**

  Задает свойство вывода. Перечень этих свойств можно найти по адресу **https://www.w3c.org/TR/xslt#output**. Ниже перечислены наиболее употребительные свойства вывода.

doctype-public	Идентификатор **PUBLIC**, используемый в объявлении **DOCTYPE**
doctype-system	Идентификатор **SYSTEM**, используемый в объявлении **DOCTYPE**
indent	Принимает значение **"yes"** или **"no"**
method	Принимает значение **"xml"**, **"html"**, **"text"** или специальное строковое значение

- **void transform(Source *from*, Result *to*)**

  Выполняет преобразование XML-документа.

javax.xml.transform.dom.DOMSource 1.4

- **DOMSource(Node *n*)**

  Создает источник данных из заданного узла. Обычно параметр ***n*** обозначает узел документа.

javax.xml.transform.stream.StreamResult 1.4

- **StreamResult(File *f*)**
- **StreamResult(OutputStream *out*)**
- **StreamResult(Writer *out*)**
- **StreamResult(String *systemID*)**

  Создают поток вывода результатов преобразования на основе указанного файла, потока вывода, потока записи или системного идентификатора (как правило, это относительный или абсолютный URL).

3.8.4. Запись XML-документов средствами StAX

В предыдущем разделе было показано, как XML-документ формируется путем записи дерева модели DOM. Но если дерево модели DOM нигде больше не используется, то такой способ оказывается не особенно эффективным. Прикладной интерфейс StAX API позволяет записывать дерево формируемого документа непосредственно в формате XML. Для этого следует создать поток записи типа `XMLStreamWriter` из потока вывода типа `OutputStream`, как показано ниже.

```
XMLOutputFactory factory = XMLOutputFactory.newInstance();
XMLStreamWriter writer = factory.createXMLStreamWriter(out);
```

Для того чтобы создать и вывести заголовок XML-документа, необходимо вызвать сначала следующий метод:

```
writer.writeStartDocument()
```

а затем метод

```
writer.writeStartElement(name);
```

Далее, для вывода атрибутов следует вызвать приведенный ниже метод.

```
writer.writeAttribute(name, value);
```

Теперь можно вывести дочерние элементы разметки, снова вызвав метод `writeStartElement()`, или записать символы, вызвав следующий метод:

```
writer.writeCharacters(text);
```

После записи всех дочерних узлов следует вызвать приведенный ниже метод, который закроет текущий элемент разметки.

```
writer.writeEndElement();
```

Чтобы записать элемент разметки без дочерних элементов (например, элемент `<img .../>`), следует вызвать следующий метод:

```
writer.writeEmptyElement(name);
```

Наконец, для завершения записи в конце документа вызывается приведенный ниже метод. Этот метод закрывает все открытые элементы разметки.

```
writer.writeEndDocument();
```

Поток записи типа `XMLStreamWriter` придется все же закрыть вручную. Ведь интерфейс `XMLStreamWriter` не расширяет интерфейс `AutoCloseable`.

Как и при подходе, предполагающем применение модели DOM и языка XSLT, в данном случае можно не особенно беспокоиться о пропуске символов в значениях атрибутов и символьных данных. Но в этом случае существует вероятность того, что XML-документ будет сформирован не совсем удачно, например, со многими корневыми узлами. Кроме того, в текущей версии прикладного интерфейса StAX API не поддерживается вывод XML-документов с отступами.

В примере программы из листинга 3.9 демонстрируется применение каждого из рассмотренных выше способов записи XML-документов.

Листинг 3.9. Исходный код из файла **write/XMLWriteTest.java**

```
1   package write;
2
3   import java.io.*;
4   import java.nio.file.*;
5   import java.util.*;
6
7   import javax.xml.parsers.*;
8   import javax.xml.stream.*;
9   import javax.xml.transform.*;
10  import javax.xml.transform.dom.*;
11  import javax.xml.transform.stream.*;
12
13  import org.w3c.dom.*;
14
15  /**
16   * этой программе демонстрируется запись XML-документа
17   * в файл. Сохраняемый файл описывает модернистский
```

```
18   * рисунок в формате SVG
19   * @version 1.12 2016-04-27
20   * @author Cay Horstmann
21   */
22  public class XMLWriteTest
23  {
24    public static void main(String[] args)
25          throws Exception
26    {
27      Document doc = newDrawing(600, 400);
28      writeDocument(doc, "drawing1.svg");
29      writeNewDrawing(600, 400, "drawing2.svg");
30    }
31
32    private static Random generator = new Random();
33
34    /**
35     * Создает новый произвольный рисунок
36     * @return Дерево DOM формируемого SVG-документа
37     */
38    public static Document newDrawing(
39          int drawingWidth, int drawingHeight)
40          throws ParserConfigurationException
41    {
42      DocumentBuilderFactory factory =
43              DocumentBuilderFactory.newInstance();
44      factory.setNamespaceAware(true);
45      DocumentBuilder builder =
46              factory.newDocumentBuilder();
47      var namespace = "http://www.w3.org/2000/svg";
48      Document doc = builder.newDocument();
49      Element svgElement =
50              doc.createElementNS(namespace, "svg");
51      doc.appendChild(svgElement);
52      svgElement.setAttribute("width",
53                              "" + drawingWidth);
54      svgElement.setAttribute("height",
55                              "" + drawingHeight);
56      int n = 10 + generator.nextInt(20);
57       for (int i = 1; i <= n; i++)
58       {
59         int x = generator.nextInt(drawingWidth);
60         int y = generator.nextInt(drawingHeight);
61         int width = generator.nextInt(drawingWidth - x);
62         int height = generator.nextInt(
63                     drawingHeight - y);
64         int r = generator.nextInt(256);
65         int g = generator.nextInt(256);
66         int b = generator.nextInt(256);
67
68         Element rectElement =
69                 doc.createElementNS(namespace, "rect");
70         rectElement.setAttribute("x", "" + x);
71         rectElement.setAttribute("y", "" + y);
72         rectElement.setAttribute("width", "" + width);
73         rectElement.setAttribute("height", "" + height);
74         rectElement.setAttribute("fill",
```

```
75          String.format("#%02x%02x%02x", r, g, b));
76          svgElement.appendChild(rectElement);
77       }
78       return doc;
79    }
80
81    /**
82     * Сохраняет документ средствами DOM/XSLT
83     */
84    public static void writeDocument(
85          Document doc, String filename)
86          throws TransformerException, IOException
87    {
88       Transformer t = TransformerFactory
89                   .newInstance().newTransformer();
90       t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
91          "http://www.w3.org/TR/2000/CR-SVG-20000802/"
92          + "DTD/svg-20000802.dtd");
93       t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC,
94                         "-//W3C//DTD SVG 20000802//EN");
95       t.setOutputProperty(OutputKeys.INDENT, "yes");
96       t.setOutputProperty(OutputKeys.METHOD, "xml");
97       t.setOutputProperty("{http://xml.apache.org/xslt}"
98                         + "indent-amount", "2");
99       t.transform(new DOMSource(doc), new StreamResult(
100         Files.newOutputStream(Paths.get(filename))));
101   }
102
103   /**
104    * Записывает SVG-документ с текущим рисунком
105    * @param writer Место назначения документа
106    * @throws IOException
107    */
108   public static void writeNewDrawing(int drawingWidth,
109         int drawingHeight, String filename)
110         throws XMLStreamException, IOException
111   {
112      XMLOutputFactory factory =
113               XMLOutputFactory.newInstance();
114      XMLStreamWriter writer =
115              factory.createXMLStreamWriter(
116      Files.newOutputStream(Paths.get(filename)));
117      writer.writeStartDocument();
118      writer.writeDTD("<!DOCTYPE svg PUBLIC"
119            + " \"-//W3C//DTD SVG 20000802//EN\" "
120            + "\"http://www.w3.org/TR/2000/"
121            + "CR-SVG-20000802/DTD/svg-20000802.dtd\">");
122      writer.writeStartElement("svg");
123      writer.writeDefaultNamespace(
124             "http://www.w3.org/2000/svg");
125      writer.writeAttribute("width", "" + drawingWidth);
126      writer.writeAttribute("height", "" + drawingHeight);
127      int n = 10 + generator.nextInt(20);
128      for (int i = 1; i <= n; i++)
129      {
130        int x = generator.nextInt(drawingWidth);
131        int y = generator.nextInt(drawingHeight);
```

```
132       int width = generator.nextInt(drawingWidth - x);
133       int height = generator.nextInt(drawingHeight - y);
134       int r = generator.nextInt(256);
135       int g = generator.nextInt(256);
136       int b = generator.nextInt(256);
137       writer.writeEmptyElement("rect");
138       writer.writeAttribute("x", "" + x);
139       writer.writeAttribute("y", "" + y);
140       writer.writeAttribute("width", "" + width);
141       writer.writeAttribute("height", "" + height);
142       writer.writeAttribute("fill",
143             String.format("#%02x%02x%02x", r, g, b));
144     }
145     writer.writeEndDocument(); // closes svg element
146   }
147 }
```

javax.xml.stream.XMLOutputFactory 6

- **static XMLOutputFactory newInstance()**

  Возвращает экземпляр класса **XMLOutputFactory**.

- **XMLStreamWriter createXMLStreamWriter(OutputStream *in*)**
- **XMLStreamWriter createXMLStreamWriter(OutputStream *in*, String *characterEncoding*)**
- **XMLStreamWriter createXMLStreamWriter(Writer *in*)**
- **XMLStreamWriter createXMLStreamWriter(Result *in*)**

  Создают поток, записывающий данные в указанный поток вывода, поток записи или результат типа JAXP.

javax.xml.stream.XMLStreamWriter 6

- **void writeStartDocument()**
- **void writeStartDocument(String *xmlVersion*)**
- **void writeStartDocument(String *encoding*, String *xmlVersion*)**

  Записывают инструкцию обработки вначале XML-документа. Следует, однако, иметь в виду, что параметр ***encoding*** указывается только для записи атрибута. Он не задает кодировку символов в выводимых данных.

- **void setDefaultNamespace(String *namespaceURI*)**
- **void setPrefix(String *prefix*, String *namespaceURI*)**

  Задают пространство имен по умолчанию или пространство имен, связанное с префиксом. Объявление действительно для текущего элемента или же для корня документа, если ни один элемент не был записан.

- **void writeStartElement(String *localName*)**
- **void writeStartElement(String *namespaceURI*, String *localName*)**

  Записывают первоначальный дескриптор, заменяя параметр ***namespaceURI*** соответствующим префиксом.

javax.xml.stream.XMLStreamWriter 6 *(окончание)*

- **void writeEndElement()**

  Закрывает текущий элемент разметки.

- **void writeEndDocument()**

  Закрывает все открытые элементы разметки.

- **void writeEmptyElement(String *localName*)**
- **void writeEmptyElement(String *namespaceURI*, String *localName*)**

  Записывают самозакрывающийся дескриптор, заменяя параметр ***namespaceURI*** соответствующим префиксом.

- **void writeAttribute(String *localName*, String *value*)**
- **void writeAttribute(String *namespaceURI*, String *localName*, String *value*)**

  Записывают атрибут для текущего элемента разметки, заменяя параметр ***namespaceURI*** соответствующим префиксом.

- **void writeCharacters(String *text*)**

  Записывает символьные данные.

- **void writeCData(String *text*)**

  Записывает раздел **CDATA**.

- **void writeDTD(String *dtd*)**

  Записывает символьную строку ***dtd***, которая должна содержать объявление **DOCTYPE**.

- **void writeComment(String *comment*)**

  Записывает комментарий.

- **void close()**

  Закрывает поток записи.

3.8.5. Пример формирования файла в формате SVG

В листинге 3.9 представлен исходный код примера программы для вывода XML-документа. Эта программа рисует картину в модернистском стиле из произвольного набора прямоугольников разного цвета (рис. 3.3). Для сохранения результатов используется формат SVG (Scalable Vector Graphics — масштабируемая векторная графика). По существу, формат SVG является разновидностью формата XML и служит для описания сложной графики в машинно-независимой форме. Дополнительные сведения об этом формате можно найти по адресу `https://www.w3c.org/Graphics/SVG`. Для просмотра файлов в формате SVG достаточно воспользоваться любым современным браузером.

В рассматриваемом здесь примере программы демонстрируются два способа формирования XML-документа: построение и сохранение дерева DOM, а также непосредственная запись XML-документа средствами прикладного интерфейса StAX API. Мы не будем вдаваться в подробности формата SVG, отсылая интересующихся за дополнительными сведениями по указанному выше адресу. Для целей рассматриваемого здесь примера достаточно знать, каким образом набор

цветных прямоугольников размечается в формате SVG. Ниже приведен пример такой разметки.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN"
    "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD
          /svg-20000802.dtd">
<svg xmlns="http://www.w3.org/2000/svg"
     width="300" height="150">
  <rect x="231" y="61" width="9" height="12"
     fill="#6e4a13"/>
  <rect x="107" y="106" width="56" height="5"
    fill="#c406be"/>
. . .
</svg>
```

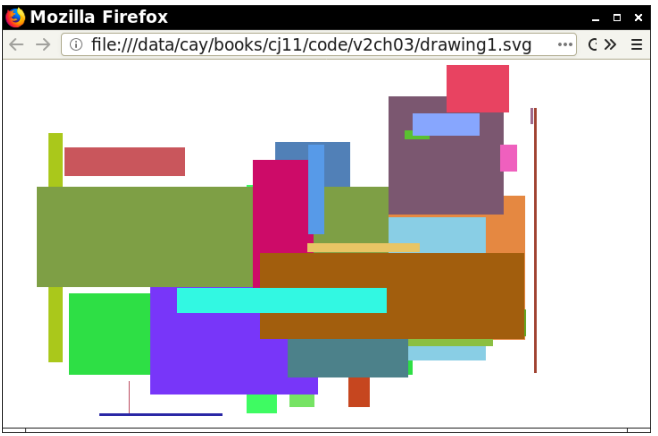


Рис. 3.3. Изображение в модернистском стиле, сохраняемое в формате SVG

Как видите, каждый прямоугольник описывается в виде узла `rect`, атрибуты которого задают координаты, ширину, высоту и цвет прямоугольника. Цвет заливки прямоугольников обозначается в виде значений основных цветов RGB в шестнадцатеричной форме.

НА ЗАМЕТКУ! В формате SVG широко применяются атрибуты. На самом деле некоторые из них имеют очень сложную структуру. В качестве примера ниже приведен элемент разметки контура.

```
<path d="M 100 100 L 300 100 L 200 300 z">
```

где **M** обозначает команду **moveto** (перейти), **L** — команду **lineto** (нарисовать линию), **z** — команду **closepath** (замкнуть контур). Вероятно, создатели формата SVG не особенно доверяли формату XML. Очевидно, что вместо таких сложных атрибутов следовало бы использовать элементы разметки в коде XML.

3.9. Преобразование XML-документов языковыми средствами XSLT

Язык преобразования XML-документов (XSLT) позволяет определять правила преобразования подобных документов в другие форматы, включая простой текст, XHTML или любую другую разновидность формата XML. Язык XSLT обычно применяется для перевода из одной машиночитаемой разновидности формата XML в другую машиночитаемую или удобочитаемую (для человека) разновидность этого формата.

Для этой цели следует создать таблицу стилей XSLT, описывающую преобразование XML-документов в какой-нибудь другой формат. Процессор XSLT сначала читает XML-документ и таблицу стилей, а затем выдает желаемый результат (рис. 3.4).

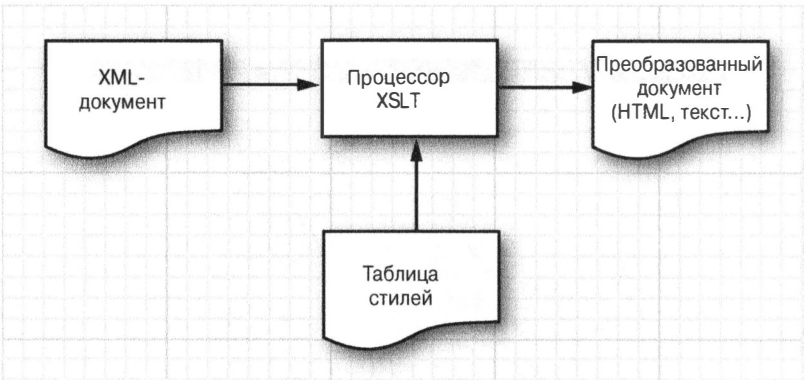


Рис. 3.4. Преобразование XML-документа языковыми средствами XMLT

Спецификация языка XSLT довольно сложна, и ее описанию посвящены целые книги. Здесь недостаточно места для описания всех языковых средств XSLT, поэтому рассмотрим лишь наглядный пример их применения. Подробнее ознакомиться с особенностями языка XSLT можно, обратившись к книге *Essential XML* Дона Бокса и др., упоминавшейся в начале этой главы, а спецификация языка XSLT доступна по адресу https://www.w3.org/TR/xslt/all/.

Допустим, XML-документ с записями о сотрудниках требуется преобразовать в HTML-документ. Ниже приведена разметка исходного XML-документа.

```
<staff>
  <employee>
    <name>Carl Cracker</name>
    <salary>75000</salary>
    <hiredate year="1987" month="12" day="15"/>
  </employee>
  <employee>
    <name>Harry Hacker</name>
    <salary>50000</salary>
    <hiredate year="1989" month="10" day="1"/>
  </employee>
  <employee>
    <name>Tony Tester</name>
    <salary>40000</salary>
```

```
    <hiredate year="1990" month="3" day="15"/>
  </employee>
</staff>
```

Из этой разметки желательно получить следующую HTML-таблицу:

```
<table border="1">
<tr>
<td>Carl Cracker</td><td>$75000.0</td><td>1987-12-15</td>
</tr>
<tr>
<td>Harry Hacker</td><td>$50000.0</td><td>1989-10-1</td>
</tr>
<tr>
<td>Tony Tester</td><td>$40000.0</td><td>1990-3-15</td>
</tr>
</table>
```

А вот как выглядит таблица стилей с шаблонами преобразования:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="html"/>
    шаблон1
    шаблон2
    . . .
</xsl:stylesheet>
```

В данном примере элемент разметки `xsl:output` содержит атрибут `method` со значением `"html"` для преобразования в формат HTML. Для данного атрибута можно также задать значения `xml` и `text`. Типичный шаблон имеет следующий вид:

```
<xsl:template match="/staff/employee">
  <tr><xsl:apply-templates/></tr>
</xsl:template>
```

Значением атрибута `match` является выражение XPath. Данный шаблон означает, что при обнаружении узла во множестве `/staff/employee`, указанном в выражении XPath, необходимо выполнить следующие действия:

1. Сформировать символьную строку с открывающим дескриптором `<tr>`.
2. Применить шаблоны при обработке дочерних узлов данного узла.
3. После обработки всех дочерних узлов сформировать символьную строку с закрывающим дескриптором `</tr>`.

Иными словами, этот шаблон заключает каждую запись о сотрудниках в дескрипторы строк HTML-таблицы.

Процессор XSLT начинает обработку XML-документа с проверки корневого элемента разметки. Если узел совпадает с одним из шаблонов, соответствующий шаблон сразу же применяется. (При совпадении с несколькими шаблонами используется шаблон с наибольшей степенью соответствия. Дополнительные сведения по данному вопросу приведены в спецификации языка XSLT по адресу

https://www.w3.org/TR/xslt/all/.) Если совпадение с шаблоном не обнаружено, процессор XSLT выполняет действие, задаваемое по умолчанию. Так, содержимое текстовых узлов по умолчанию включается в выводимый результат, а для элементов разметки выводимый результат не формируется, но продолжается обработка дочерних элементов.

В качестве примера ниже приведен шаблон преобразования узлов `name` из XML-документа с данными о сотрудниках.

```
<xsl:template match="/staff/employee/name">
  <td><xsl:apply-templates/></td>
</xsl:template>
```

Как видите, шаблон формирует дескрипторы `<td>...</td>` и предписывает процессору XSLT рекурсивно обойти дочерние узлы элемента разметки `name`. У этого элемента имеется только один дочерний текстовый узел. Когда процессор обходит узел, он возвращает его текстовое содержимое, если, конечно, отсутствуют другие совпавшие шаблоны.

Для копирования значений атрибутов в выводимый результат придется задать более сложный шаблон:

```
<xsl:template match="/staff/employee/hiredate">
  <td><xsl:value-of select="@year"/>-<xsl:value-of
  select="@month"/>-<xsl:value-of select="@day"/></td>
</xsl:template>
```

При обработке узла `hiredate` по этому шаблону будут сформированы перечисленные ниже элементы разметки таблицы.

1. Дескриптор `<td>`.
2. Значение атрибута `year`.
3. Дефис.
4. Значение атрибута `month`.
5. Дефис.
6. Значение атрибута `day`.
7. Дескриптор `</td>`.

Оператор `xsl:value-of` вычисляет строковое значение множества узлов, указываемого с помощью значения XPath атрибута `select`. В этом случае путь определяется относительно текущего узла. Множество узлов преобразуется в символьную строку путем сцепления строковых значений из всех узлов. Строковым значением атрибута является его значение, строковым значением текстового узла — его содержимое, а строковым значением элемента разметки — сцепление строковых значений его дочерних узлов (но не атрибутов).

В листинге 3.10 приведена таблица стилей для преобразования XML-документа с записями о сотрудниках в HTML-таблицу.

Листинг 3.10. Исходный код из файла **transform/makehtml.xsl**

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:output method="html"/>

  <xsl:template match="/staff">
    <table border="1"><xsl:apply-templates/></table>
  </xsl:template>

  <xsl:template match="/staff/employee">
    <tr><xsl:apply-templates/></tr>
  </xsl:template>

  <xsl:template match="/staff/employee/name">
    <td><xsl:apply-templates/></td>
  </xsl:template>

  <xsl:template match="/staff/employee/salary">
    <td>$<xsl:apply-templates/></td>
  </xsl:template>

  <xsl:template match="/staff/employee/hiredate">
    <td><xsl:value-of select="@year"/>-<xsl:value-of
    select="@month"/>-<xsl:value-of select="@day"/></td>
  </xsl:template>

</xsl:stylesheet>
```

В листинге 3.11 приведены шаблоны для различных преобразований того же самого XML-документа в обычный текст в знакомом уже формате файла свойств:

```
employee.1.name=Carl Cracker
employee.1.salary=75000.0
employee.1.hiredate=1987-12-15
employee.2.name=Harry Hacker
employee.2.salary=50000.0
employee.2.hiredate=1989-10-1
employee.3.name=Tony Tester
employee.3.salary=40000.0
employee.3.hiredate=1990-3-15
```

Листинг 3.11. Исходный код из файла **transform/makeprop.xsl**

```
<?xml version="1.0"?>

<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:output method="text" omit-xml-declaration="yes"/>

```

```
9    <xsl:template match="/staff/employee">
10 employee.<xsl:value-of select="position()"
11 />.name=<xsl:value-of select="name/text()"/>
12 employee.<xsl:value-of select="position()"
13 />.salary=<xsl:value-of select="salary/text()"/>
14 employee.<xsl:value-of select="position()"
15 />.hiredate=<xsl:value-of select="hiredate/@year"
16 />-<xsl:value-of select="hiredate/@month"
17 />-<xsl:value-of select="hiredate/@day"/>
18   </xsl:template>
19
20 </xsl:stylesheet>
```

В данном примере используется функция `position()`, которая выдает расположение текущего узла относительно родительского. Чтобы получить выводимый результат в совершенно другом виде, достаточно внести соответствующие изменения в таблицу стилей. Таким образом, XML-документ можно благополучно применять для представления данных, не особенно заботясь, в каком именно формате они требуются в прикладной программе. Для формирования данных в нужном формате достаточно воспользоваться средствами XSLT.

Преобразования средствами XSLT совсем не трудно организовать на платформе Java. С этой целью следует создать сначала отдельную фабрику преобразователей для каждой таблицы стилей, а затем получить объект типа `Transformer` и передать ему преобразуемые данные, как показано ниже.

```
var styleSheet = new File(filename);
var styleSource = new StreamSource(styleSheet);
Transformer t = TransformerFactory.newInstance()
              .newTransformer(styleSource);
t.transform(source, result);
```

Параметры, передаваемые методу `transform()`, представляют собой экземпляры классов, реализующих интерфейсы `Source` и `Result`. Так, у интерфейса `Source` имеются реализации источника данных в следующих классах:

```
DOMSource
SAXSource
StAXSource
StreamSource
```

Потоковый источник данных типа `StreamSource` можно создать из файла, потока ввода, потока чтения или URL, а источник данных типа `DOMSource` — из узла дерева DOM. Так, в примере программы из листинга 3.9 в предыдущем разделе выполнялось следующее тождественное преобразование:

```
t.transform(new DOMSource(doc), result);
```

В рассматриваемом здесь примере программы применяется другой, более интересный подход. Вместо уже существующего XML-файла создается поток, читающий данные в формате XML, имитируя их SAX-анализ с инициированием соответствующих SAX-событий. По существу, поток чтения данных формата XML выполняет чтение из однородного входного файла со следующим содержимым:

```
Carl Cracker|75000.0|1987|12|15
Harry Hacker|50000.0|1989|10|1
Tony Tester|40000.0|1990|3|15
```

По мере обработки вводимых данных формата XML в потоке чтения инициируются SAX-события. Ниже представлен фрагмент исходного кода метода `parse()` из класса `EmployeeReader`, реализующего интерфейс `XMLReader`.

```
var attributes = new AttributesImpl();
handler.startDocument();
handler.startElement("", "staff", "staff", attributes);
while ((line = in.readLine()) != null)
{
  handler.startElement("", "employee", "employee",
                       attributes);
  var tokenizer = new StringTokenizer(line, "|");
  handler.startElement("", "name", "name", attributes);
  String s = tokenizer.nextToken();
  handler.characters(s.toCharArray(), 0, s.length());
  handler.endElement("", "name", "name");
  . . .
  handler.endElement("", "employee", "employee");
}
handler.endElement("", rootElement, rootElement);
handler.endDocument();
```

Источник типа `SAXSource` для преобразования данных создается из потока чтения XML-данных следующим образом:

```
t.transform(new SAXSource(new EmployeeReader(),
  new InputSource(new FileInputStream(filename))), result);
```

Такой искусный прием как нельзя лучше подходит для преобразования унаследованных данных в формат XML. Безусловно, для большинства приложений XSLT вводимые данные уже находятся в формате XML, что позволяет просто вызвать метод `transform()` для объекта типа `SAXSource`:

```
t.transform(new StreamSource(file), result);
```

Результатом такого преобразования оказывается объект одного из следующих трех классов, реализующих интерфейс `Result` в библиотеке Java:

```
DOMResult
SAXResult
StreamResult
```

Для сохранения результата в виде древовидной структуры DOM используется объект типа `DocumentBuilder`. С его помощью генерируется новый узел документа, заключаемый в оболочку типа `DOMResult`, как показано ниже.

```
Document doc = builder.newDocument();
t.transform(source, new DOMResult(doc));
```

Наконец, для вывода полученного результата в файл используется объект типа `StreamResult`:

```
t.transform(source, new StreamResult(file));
```

В листинге 3.12 приведен весь исходный код рассмотренного здесь примера программы.

Листинг 3.12. Исходный код из файла **transform/TransformTest.java**

```
1  package transform;
2
3  import java.io.*;
4  import java.nio.file.*;
5  import java.util.*;
6  import javax.xml.transform.*;
7  import javax.xml.transform.sax.*;
8  import javax.xml.transform.stream.*;
9  import org.xml.sax.*;
10 import org.xml.sax.helpers.*;
11
12 /**
13  * В этой программе демонстрируются преобразования
14  * языковыми средствами XSLT. Преобразования
15  * выполняются над записями о сотрудниках из файла
16  * employee.dat в формат XML. Для этого следует
17  * указать таблицу стилей в командной строке,
18  * например, следующим образом:
19  * java TransformTest transform/makeprop.xsl
20  * @version 1.04 2018-04-10
21  * @author Cay Horstmann
22  */
23 public class TransformTest
24 {
25   public static void main(String[] args)
26         throws Exception
27   {
28     Path path;
29     if (args.length > 0) path = Paths.get(args[0]);
30     else path = Paths.get("transform", "makehtml.xsl");
31     try (InputStream styleIn =
32          Files.newInputStream(path))
33     {
34       var styleSource = new StreamSource(styleIn);
35
36       Transformer t = TransformerFactory.newInstance()
37                     .newTransformer(styleSource);
38       t.setOutputProperty(OutputKeys.INDENT, "yes");
39       t.setOutputProperty(OutputKeys.METHOD, "xml");
40       t.setOutputProperty("{http://xml.apache.org/xslt}"
41                           + "indent-amount", "2");
42
43       try (InputStream docIn = Files.newInputStream(
44            Paths.get("transform", "employee.dat")))
45       {
46         t.transform(new SAXSource(new EmployeeReader(),
47                     new InputSource(docIn)),
48                     new StreamResult(System.out));
49       }
50     }
51   }
52 }
53
54 /**
```

```
55   * Этот класс читает однородный файл employee.dat и
56   * уведомляет о событиях в SAX-анализаторе, как будто
57   * он сам выполнил синтаксический анализ XML-документа
58   */
59  class EmployeeReader implements XMLReader
60  {
61    private ContentHandler handler;
62
63    public void parse(InputSource source)
64           throws IOException, SAXException
65    {
66      InputStream stream = source.getByteStream();
67      var in = new BufferedReader(
68               new InputStreamReader(stream));
69      String rootElement = "staff";
70      var atts = new AttributesImpl();
71
72      if (handler == null)
73        throw new SAXException("No content handler");
74
75      handler.startDocument();
76      handler.startElement("", rootElement,
77                           rootElement, atts);
78      String line;
79      while ((line = in.readLine()) != null)
80      {
81        handler.startElement("", "employee",
82                             "employee", atts);
83        var t = new StringTokenizer(line, "|");
84
85        handler.startElement("", "name", "name", atts);
86        String s = t.nextToken();
87        handler.characters(s.toCharArray(), 0,
88                           s.length());
89        handler.endElement("", "name", "name");
90
91        handler.startElement("", "salary", "salary",
92                             atts);
93        s = t.nextToken();
94        handler.characters(s.toCharArray(), 0,
95                           s.length());
96        handler.endElement("", "salary", "salary");
97
98        atts.addAttribute("", "year", "year", "CDATA",
99                          t.nextToken());
100       atts.addAttribute("", "month", "month", "CDATA",
101                         t.nextToken());
102       atts.addAttribute("", "day", "day", "CDATA",
103                         t.nextToken());
104       handler.startElement("", "hiredate", "hiredate",
105                            atts);
106       handler.endElement("", "hiredate", "hiredate");
107       atts.clear();
108
109       handler.endElement("", "employee", "employee");
110     }
```

```
111
112       handler.endElement("", rootElement, rootElement);
113       handler.endDocument();
114    }
115
116    public void setContentHandler(ContentHandler newValue)
117    {
118       handler = newValue;
119    }
120
121    public ContentHandler getContentHandler()
122    {
123       return handler;
124    }
125
126    // следующие методы являются всего лишь
127    // холостыми реализациями
128    public void parse(String systemId)
129           throws IOException, SAXException {}
130    public void setErrorHandler(ErrorHandler handler) {}
131    public ErrorHandler getErrorHandler() { return null; }
132    public void setDTDHandler(DTDHandler handler) {}
133    public DTDHandler getDTDHandler() { return null; }
134    public void setEntityResolver(
135           EntityResolver resolver) {}
136    public EntityResolver getEntityResolver()
137           { return null; }
138    public void setProperty(String name, Object value) {}
139    public Object getProperty(String name)
140           { return null; }
141    public void setFeature(String name, boolean value) {}
142    public boolean getFeature(String name)
143           { return false; }
144 }
```

javax.xml.transform.TransformerFactory 1.4

- **Transformer newTransformer(Source *styleSheet*)**

  Возвращает экземпляр класса **Transformer**, считывающий таблицу стилей из указанного источника.

javax.xml.transform.stream.StreamSource 1.4

- **StreamSource(File *f*)**
- **StreamSource(InputStream *in*)**
- **StreamSource(Reader *in*)**
- **StreamSource(String *systemID*)**

  Создают потоковый источник данных из указанного файла, потока ввода, потока чтения или системного идентификатора (обычно это относительный или абсолютный URL).

javax.xml.transform.sax.SAXSource 1.4

- **SAXSource(XMLReader *reader*, InputSource *source*)**

  Создает SAX-источник, получающий вводимые данные из указанного источника, используя заданный поток чтения для синтаксического анализа данных.

org.xml.sax.XMLReader 1.4

- **void setContentHandler(ContentHandler *handler*)**

  Устанавливает обработчик, который уведомляется о событиях, наступающих при синтаксическом анализе вводимых данных.

- **void parse(InputSource *source*)**

  Анализирует данные, вводимые из указанного источника, и передает обработчику содержимого события, наступающие при синтаксическом анализе этих данных.

javax.xml.transform.dom.DOMResult 1.4

- **DOMResult(Node *n*)**

  Создает источник данных из заданного узла. Обычно в качестве параметра ***n*** указывается узел документа.

org.xml.sax.helpers.AttributesImpl 1.4

- **void addAttribute(String *uri*, String *lname*, String *qname*, String *type*, String *value*)**

  Вводит атрибут в коллекцию атрибутов. В качестве параметра ***lname*** указывается локальное имя без префикса, а в качестве параметра ***qname*** — уточненное имя с префиксом. Параметр ***type*** принимает одно из следующих строковых значений: **"CDATA"**, **"ID"**, **"IDREF"**, **"IDREFS"**, **"NMTOKEN"**, **"NMTOKENS"**, **"ENTITY"**, **"ENTITIES"** или **"NOTATION"**.

- **void clear()**

  Удаляет все атрибуты из данной коллекции.

Этим примером завершается обсуждение особенностей поддержки XML в библиотеке Java. Теперь у вас должно сложиться ясное представление о возможностях XML, включая автоматизированный синтаксический анализ и проверку достоверности, а также эффективный механизм преобразования XML-документов. Естественно, что всю эту технологию вам удастся поставить себе на службу лишь в том случае, если вы тщательно разработаете свои форматы XML. Для этого ваши форматы XML должны удовлетворять всем насущным производственным

потребностям, сохранять устойчивость с течением времени, а ваши деловые партнеры быть готовыми принимать от вас XML-документы. Решение всех этих вопросов может оказаться гораздо сложнее, чем умелое обращение с синтаксическими анализаторами, определениями DTD или преобразованиями, выполняемыми средствами XSLT.

В следующей главе будут обсуждаться вопросы сетевого программирования на платформе Java. Сначала мы рассмотрим основные положения о сетевых сокетах, а затем перейдем к высокоуровневым протоколам для организации электронной почты и Всемирной паутины.

ГЛАВА 4

Работа в сети

В этой главе...

- Подключение к серверу
- Реализация серверов
- Получение данных из Интернета
- HTTP-клиент
- Отправка электронной почты

Эта глава начинается с описания основных понятий для работы в сети, а затем в ней рассматриваются примеры написания программ на Java, позволяющих устанавливать соединения с серверами. Из нее вы узнаете, как осуществляется реализация сетевых клиентов и серверов. А завершается глава рассмотрением вопросов передачи почтовых сообщений из программы на Java и сбора данных с веб-сервера.

4.1. Подключение к серверу

В последующих разделах сначала рассматривается подключение к серверу вручную с помощью утилиты `telnet`, а затем автоматическое подключение из программы на Java.

4.1.1. Применение утилиты `telnet`

Утилита `telnet` служит отличным инструментальным средством для отладки сетевых программ. Она должна запускаться из командной строки по команде **`telnet`**.

НА ЗАМЕТКУ! В Windows утилиту **telnet** необходимо активизировать. С этой целью откройте панель управления, перейдите в раздел Программы, щелкните на ссылке Добавление или удаление компонентов Windows и установите флажок Клиент Telnet. Следует также иметь в виду, что брандмауэр Windows блокирует некоторые сетевые порты, которые будут использоваться в примерах программ из этой главы. Чтобы разблокировать эти порты, вы должны обладать полномочиями администратора.

Утилитой `telnet` можно пользоваться не только для соединения с удаленным компьютером. С ее помощью можно также взаимодействовать с различными сетевыми службами. Ниже приводится один из примеров необычного использования этой утилиты. Для этого введите в командной строке следующую команду:

```
telnet time-a.nist.gov 13
```

На рис. 4.1 приведен пример ответной реакции сервера, которая в режиме командной строки будет иметь следующий вид:

```
54276 07-06-25 21:37:31 50 0 0 659.0 UTC(NIST) *
```

Рис. 4.1. Результат, получаемый из службы учета времени дня

Что же в действительности произошло? Утилита `telnet` подключилась к серверу службы учета времени дня, который работает на большинстве компьютеров под управлением операционной системы UNIX. Указанный в этом примере сервер находится в Национальном институте стандартов и технологий США (National Institute of Standards and Technology). Его системное время синхронизировано с цезиевыми атомными часами. (Безусловно, полученное значение текущего времени будет не совсем точным из-за задержек, связанных с передачей данных по сети.) По принятым правилам сервер службы времени всегда связан с портом `13`.

НА ЗАМЕТКУ! В сетевой терминологии *порт* — это не какое-то конкретное физическое устройство, а абстрактное понятие, упрощающее представление о соединении сервера с клиентом (рис. 4.2).

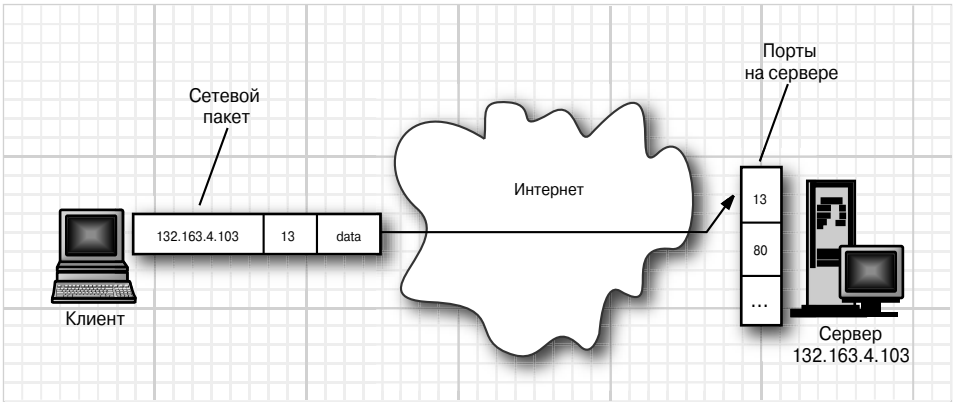


Рис. 4.2. Схема соединения клиента с сервером через конкретный порт

Программное обеспечение сервера постоянно работает на удаленном компьютере и ожидает поступления сетевого трафика через порт `13`. При получении операционной системой на удаленном компьютере сетевого пакета с запросом на подключение к порту `13` на сервере активизируется соответствующий процесс и устанавливается соединение. Такое соединение может быть прервано одним из его участников.

Когда сеанс связи с сервером через порт `13` начинается по команде `telnet` с параметром `time-a.nist.gov`, сетевое программное обеспечение преобразует строку `"time-a.nist.gov"` в IP-адрес `129.6.15.28`. Затем оно посылает по этому адресу запрос на соединение с удаленным компьютером через порт `13`. После установления соединения программа на удаленном компьютере передает обратно строку с данными, а затем разрывает соединение. Разумеется, клиенты и серверы могут вести и более сложные диалоги до разрыва соединения.

Проведем еще один, более интересный эксперимент. С этой целью выполните следующие действия.

1. Введите в режиме командной строки команду

   ```
   telnet horstmann.com 80
   ```

2. Затем аккуратно и точно введите следующие строки, *дважды* нажав клавишу <Enter> в конце:

   ```
   GET / HTTP/1.1
   Host: horstmann.com
   пустая строка
   ```

На рис. 4.3 показана ответная реакция сервера в окне утилиты `telnet`. Она имеет уже знакомый вам вид страницы текста в формате HTML, а именно начальной страницы веб-сайта Кея Хорстманна. Именно так обычный веб-браузер

получает искомые веб-страницы. Для запроса веб-страниц на сервере он применяет сетевой протокол HTTP. Разумеется, браузер отображает данные в намного более удобном для чтения виде, чем формат HTML.

Рис. 4.3. Доступ к HTTP-порту с помощью утилиты **`telnet`**

НА ЗАМЕТКУ! Пару "ключ–значение" **`Host: horstmann.com`** требуется указывать для подключения к веб-серверу, на котором под одним и тем же IP-адресом размещаются разные домены. Ее можно не указывать, если на веб-сервере размещается единственный домен.

4.1.2. Подключение к серверу из программы на Java

В первом примере сетевой программы, исходный код которой приведен в листинге 4.1, выполняются те же действия, что и при использовании утилиты `telnet`. Она устанавливает соединение с сервером через порт и выводит получаемые в ответ данные.

Листинг 4.1. Исходный код из файла **`socket/SocketTest.java`**

```
1  package socket;
2
3  import java.io.*;
4  import java.net.*;
5  import java.nio.charset.*;
6  import java.util.*;
7
8  /**
9   * В этой программе устанавливается сокетное соединение
10  * с атомными часами в г. Боулдере, шт. Колорадо и
11  * выводится время, передаваемое из сервера
12  * @version 1.22 2018-03-17
13  * @author Cay Horstmann
14  */
```

```
15 public class SocketTest
16 {
17   public static void main(String[] args)
18         throws IOException
19   {
20     try (var s = new Socket("time-a.nist.gov", 13);
21     var in = new Scanner(s.getInputStream(),
22                          StandardCharsets.UTF_8))
23     {
24       while (in.hasNextLine())
25       {
26         String line = in.nextLine();
27         System.out.println(line);
28       }
29     }
30   }
31 }
```

В данной программе наибольший интерес представляют следующие две строки кода:

```
Socket s = new Socket("time-a.nist.gov", 13);
Scanner in = new Scanner(s.getInputStream(), "UTF-8"))
```

В первой строке кода открывается сокет. *Сокет* — это абстрактное понятие, обозначающее возможность для программ устанавливать соединения для обмена данными по сети. Конструктору объекта сокета передается адрес удаленного сервера и номер порта. Если установить соединение не удается, генерируется исключение типа `UnknownHostException`, а при возникновении каких-нибудь других затруднений — исключение типа `IOException`. Класс `UnknownHostException` является подклассом, производным от класса `IOException`, поэтому в данном простом примере обрабатывается только исключение из суперкласса.

После открытия сокета метод `getInputStream()` из класса `java.net.Socket` возвращает объект типа `InputStream`, который можно использовать как любой другой поток ввода. Получив поток ввода, рассматриваемая здесь программа приступает к выводу каждой введенной символьной строки в стандартный поток вывода. Этот процесс продолжается до тех пор, пока не завершится поток ввода или не разорвется соединение с сервером.

Данная программа может взаимодействовать только с очень простыми серверами, например со службой учета текущего времени. В более сложных случаях клиент посылает серверу запрос на получение данных, а сервер может поддерживать установленное соединение в течение некоторого времени после отправки ответа на запрос. Примеры реализации подобного поведения представлены далее в этой главе.

Класс `Socket` очень удобен для работы в сети, поскольку он скрывает все сложности и подробности установления сетевого соединения и передачи данных по сети, реализуемые средствами библиотеки Java. Пакет `java.net`, по существу, предоставляет тот же самый программный интерфейс, который используется для работы с файлами.

НА ЗАМЕТКУ! Здесь рассматривается только сетевой протокол TCP (Transmission Control Protocol — протокол управления передачей). На платформе Java поддерживается также протокол UDP (User Datagram Protocol — протокол пользовательских дейтаграмм), который может служить для отправки пакетов (называемых иначе *дейтаграммами*) с гораздо меньшими издержками, чем по протоколу TCP. Недостаток такого способа обмена данными по сети заключается в том, что пакеты необязательно доставлять получателю в последовательном порядке, и они вообще могут быть потеряны. Получатель сам должен позаботиться о том, чтобы пакеты были организованы в определенном порядке, а кроме того, он должен сам запрашивать повторно передачу отсутствующих пакетов. Протокол UDP хорошо подходит для тех приложений, которые могут обходиться без отсутствующих пакетов, например, для организации аудио- и видеопотоков или продолжительных измерений.

java.net.Socket 1.0

- **Socket(String *host*, int *port*)**
  Создает сокет для соединения с указанным хостом или портом.
- **InputStream getInputStream()**
- **OutputStream getOutputStream()**
  Получают поток ввода для чтения данных из сокета или поток вывода для записи данных в сокет.

4.1.3. Время ожидания для сокетов

Чтение данных из сокета продолжается до тех пор, пока данные доступны. Если хост (т.е. сетевой узел) недоступен, прикладная программа будет ожидать очень долго, и все будет зависеть от того, когда операционная система, под управлением которой работает компьютер, определит момент завершения времени ожидания.

Для конкретной прикладной программы можно самостоятельно определить наиболее подходящую величину времени ожидания для сокета, а затем вызвать метод `setSoTimeout()`, чтобы установить эту величину в миллисекундах. В приведенном ниже фрагменте кода показано, как это делается.

```
var s = new Socket(. . .);
// истечение времени ожидания через 10 секунд:
s.setSoTimeout(10000);
```

Если величина времени ожидания была задана для сокета, то при выполнении всех последующих операций чтения и записи данных будет генерироваться исключение типа `SocketTimeoutException` по истечении времени ожидания до фактического завершения текущей операции. Но это исключение можно перехватить, чтобы отреагировать на данное событие надлежащим образом, как показано ниже.

```
try
{
  InputStream in = s.getInputStream();
    // читать данные из потока ввода in
  . . .
}
```

```
catch (InterruptedIOException exception)
{
  отреагировать на истечение времени ожидания
}
```

Что касается времени ожидания для сокетов, то остается еще одно затруднение, которое придется каким-то образом разрешить. Так, приведенный ниже конструктор может установить блокировку в течение неопределенного периода времени до тех пор, пока не будет установлено первоначальное соединение с хостом.

```
Socket(String host, int port)
```

Это затруднение можно преодолеть, если сначала создать несоединяемый сокет, а затем установить соединение с ним, задав время ожидания:

```
Socket s = new Socket();
s.connect(new InetSocketAddress(host, port), timeout);
```

Если же пользователям требуется предоставить возможность прерывать соединение с сокетом в любой момент, то далее, в разделе 4.2.4 поясняется, как этого добиться.

java.net.Socket 1.0

- **Socket() 1.1**
  Создает сокет, который еще не соединен в данный момент времени.
- **void connect(SocketAddress *address*) 1.4**
  Соединяет данный сокет по указанному адресу.
- **void connect(SocketAddress *address*, int *timeoutInMilliseconds*) 1.4**
  Соединяет данный сокет по указанному адресу или осуществляет возврат, если заданный промежуток времени истек.
- **void setSoTimeout(int *timeoutInMilliseconds*) 1.1**
  Задает время ожидания для чтения запросов в данном сокете. По истечении времени ожидания возникает исключение типа **InterruptedIOException**.
- **boolean isConnected() 1.4**
  Возвращает логическое значение **true**, если установлено соединение с сокетом.
- **boolean isClosed() 1.4**
  Возвращает логическое значение **true**, если разорвано соединение с сокетом.

4.1.4. Межсетевые адреса

Как правило, нет особой нужды беспокоиться о межсетевых адресах в Интернете — числовых адресах хостов, состоящих из четырех байтов (или из шестнадцати байтов — по протоколу IPv6), как, например, `129.6.15.28`. Но если требуется выполнить взаимное преобразование имен хостов и межсетевых адресов, то для этой цели можно воспользоваться классом `InetAddress`.

В пакете `java.net` поддерживаются межсетевые адреса по протоколу IPv6, при условии, что их поддержка обеспечивается и со стороны операционной

системы хоста. В частности, статический метод `getByName()` возвращает объект типа `InetAddress` для хоста. Например, в следующей строке кода возвращается объект типа `InetAddress`, инкапсулирующий последовательность из четырех байтов `129.6.15.28`:

```
InetAddress address = InetAddress.getByName("time-a.nist.gov");
```

Чтобы получить байты межсетевого адреса, достаточно вызвать метод `getAddress()` следующим образом:

```
byte[] addressBytes = address.getAddress();
```

Имена некоторых хостов с большим объемом трафика соответствуют нескольким межсетевым адресам, что объясняется попыткой сбалансировать нагрузку. Так, на момент написания данной книги имя хоста `google.com` соответствовало двенадцати различным сетевым адресам. Один из них выбирается случайным образом во время доступа к хосту. Получить межсетевые адреса всех хостов можно, вызвав метод `getAllByName()`:

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

Наконец, иногда требуется адрес локального хоста. Если вы просто запросите адрес локального хоста, указав `localhost`, то неизменно получите в ответ локальный петлевой адрес `127.0.0.1`, которым другие не смогут воспользоваться для подключения к вашему компьютеру. Вместо этого вызовите метод `getLocalHost()`, чтобы получить адрес вашего локального хоста, как показано ниже.

```
InetAddress address = InetAddress.getLocalHost();
```

В листинге 4.2 приведен пример простой программы, выводящей межсетевой адрес локального хоста, если не указать дополнительные параметры в командной строке, или же все межсетевые адреса другого хоста, если указать имя хоста в командной строке, как в следующем примере:

```
java inetAddress/InetAddressTest www.horstmann.com
```

Листинг 4.2. Исходный код из файла **`inetAddress/InetAddressTest.java`**

```
1  package inetAddress;
2
3  import java.io.*;
4  import java.net.*;
5  /**
6   * В этой программе демонстрируется применение
7   * класса InetAddress. В качестве аргумента в командной
8   * строке следует указать имя хоста или запустить
9   * программу без аргументов, чтобы получить в ответ
10  * адрес локального хоста
11  * @version 1.02 2012-06-05
12  * @author Cay Horstmann
13  */
14 public class InetAddressTest
15 {
```

```
16   public static void main(String[] args)
17          throws IOException
18   {
19     if (args.length > 0)
20     {
21       String host = args[0];
22       InetAddress[] addresses =
23                      InetAddress.getAllByName(host);
24       for (InetAddress a : addresses)
25         System.out.println(a);
26     }
27     else
28     {
29       InetAddress localHostAddress =
30                    InetAddress.getLocalHost();
31       System.out.println(localHostAddress);
32     }
33   }
34 }
```

java.net.InetAddress 1.0

- **static InetAddress getByName(String *host*)**
- **static InetAddress[] getAllByName(String *host*)**

  Конструируют объект типа **InetAddress** или массив всех межсетевых адресов для заданного имени хоста.
- **static InetAddress getLocalHost()**

  Конструирует объект типа **InetAddress** для локального хоста.
- **byte[] getAddress()**

  Возвращает массив байтов, содержащий числовой адрес.
- **String getHostAddress()**

  Возвращает адрес хоста в виде символьной строки с десятичными числами, разделенными точками, например **"132.163.4.102"**.
- **String getHostName()**

  Возвращает имя хоста.

4.2. Реализация серверов

В предыдущем разделе были рассмотрены особенности реализации элементарного сетевого клиента, способного получать данные из сети вообще и Интернета в частности. Теперь перейдем к обсуждению реализации простого сервера, способного посылать данные клиентам.

4.2.1. Сокеты сервера

После запуска серверная программа переходит в режим ожидания от клиентов подключения к портам сервера. Для рассматриваемого здесь примера выбран

номер порта 8189, который не используется ни одним из стандартных устройств. В следующей строке кода создается сервер с контролируемым портом 8189:

```
var s = new ServerSocket(8189);
```

В приведенной ниже строке кода серверной программе предписывается ожидать подключения клиентов к заданному порту.

```
Socket incoming = s.accept();
```

Как только какой-нибудь клиент подключится к данному порту, отправив по сети запрос на сервер, метод `accept()` возвратит объект типа `Socket`, представляющий установленное соединение. Этот объект можно использовать для чтения и записи данных в потоки ввода-вывода, как показано в приведенном ниже фрагменте кода.

```
InputStream inStream = incoming.getInputStream();
OutputStream outStream = incoming.getOutputStream();
```

Все данные, направляемые в поток вывода серверной программы, поступают в поток ввода клиентской программы. А все данные, направляемые в поток вывода из клиентской программы, поступают в поток ввода серверной программы. Во всех примерах, приведенных в этой главе, обмен текстовыми данными осуществляется через сокеты. Поэтому соответствующие потоки ввода-вывода через сокет преобразуются в потоки сканирования (типа `Scanner`) и записи (типа `Writer`) следующим образом:

```
var in = new Scanner(inStream, "UTF-8");
var out = new PrintWriter(new OutputStreamWriter(
              outStream, "UTF-8"),
              true /* автоматическая очистка */);
```

Допустим, клиентская программа посылает следующее приветствие:

```
out.println("Hello! Enter BYE to exit.");
```

Если для подключения к серверной программе через порт 8189 используется утилита `telnet`, это приветствие отображается на экране терминала.

В рассматриваемой здесь простой серверной программе вводимые данные, отправленные клиентской программой, считываются построчно и посылаются обратно клиентской программе в режиме эхопередачи, как показано в приведенном ниже фрагменте кода. Этим наглядно демонстрируется получение данных от клиентской программы. Настоящая серверная программа должна обработать полученные данные и выдать соответствующий ответ.

```
String line = in.nextLine();
out.println("Echo: " + line);
if (line.trim().equals("BYE")) done = true;
```

По завершении сеанса связи открытый сокет закрывается следующим образом:

```
incoming.close();
```

Вот, собственно, и все, что делает данная программа. Любая серверная программа, например, веб-сервер, работающий по протоколу HTTP, выполняет аналогичный цикл следующих действий.

1. Получение из потока ввода входящих данных запроса на конкретную информацию от клиентской программы.
2. Расшифровка клиентского запроса.
3. Сбор информации, запрашиваемой клиентом.
4. Передача обнаруженной информации клиентской программе через поток вывода исходящих данных.

В листинге 4.3 приведен весь исходный код описанного выше примера серверной программы.

Листинг 4.3. Исходный код из файла **server/EchoServer.java**

```
1  package server;
2
3  import java.io.*;
4  import java.net.*;
5  import java.nio.charset.*;
6  import java.util.*;
7
8  /**
9   * В этой программе реализуется простой сервер,
10  * прослушивающий порт 8189 и посылающий обратно
11  * клиенту все полученные от него данные
12  * client input.
13  * @version 1.22 2018-03-17
14  * @author Cay Horstmann
15  */
16 public class EchoServer
17 {
18   public static void main(String[] args)
19          throws IOException
20   {
21     // установить сокет на стороне сервера
22     try (var s = new ServerSocket(8189))
23     {
24       // ожидать подключения клиента
25       try (Socket incoming = s.accept())
26       {
27         InputStream inStream =
28                     incoming.getInputStream();
29         OutputStream outStream =
30                      incoming.getOutputStream();
31
32         try (var in = new Scanner(inStream,
33                       StandardCharsets.UTF_8))
34         {
35           var out = new PrintWriter(
36               new OutputStreamWriter(
37                   outStream, StandardCharsets.UTF_8),
38                   true /* автоматическая очистка */);
39
40           out.println("Hello! Enter BYE to exit.");
41
```

```
42            // передать обратно данные,
43            // полученные от клиента
44            var done = false;
45            while (!done && in.hasNextLine())
46            {
47              String line = in.nextLine();
48              out.println("Echo: " + line);
49              if (line.trim().equals("BYE")) done = true;
50            }
51          }
52        }
53      }
54    }
55 }
```

Для проверки работоспособности данной серверной программы ее нужно скомпилировать и запустить. Затем необходимо подключиться с помощью утилиты `telnet` к локальному серверу `localhost` (или по IP-адресу `127.0.0.1`) через порт `8189`. Если ваш компьютер непосредственно подключен к Интернету, любой пользователь может получить доступ к данной серверной программе, если ему известен IP-адрес и номер порта. При подключении через этот порт будет получено следующее сообщение (рис. 4.4):

```
Hello! Enter BYE to exit.[1]
```

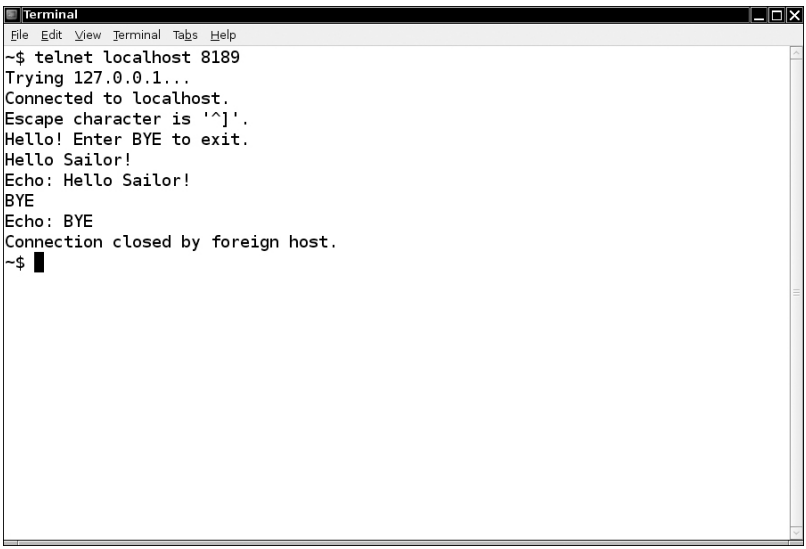


Рис. 4.4. Сеанс связи с сервером, передающим обратно данные, полученные от клиента

Введите любую фразу и понаблюдайте за тем, как она будет получена обратно в том же самом виде. Для отключения от сервера введите **BYE** (все символы в верхнем регистре). В итоге завершится и серверная программа.

---

[1] Привет! Введите BYE (Пока), чтобы выйти из программы.

java.net.ServerSocket 1.0

- **ServerSocket(int *port*)**
  Создает сокет на стороне сервера, контролирующего указанный порт.
- **Socket accept()**
  Ожидает соединения. Этот метод блокирует (т.е. переводит в режим ожидания) текущий поток до тех пор, пока не будет установлено соединение. Возвращает объект типа **Socket**, через который программа может взаимодействовать с подключаемым клиентом.
- **void close()**
  Закрывает сокет на стороне сервера.

4.2.2. Обслуживание многих клиентов

В предыдущем простом примере серверной программы не предусмотрена возможность одновременного подключения сразу нескольких клиентских программ. Обычно серверная программа работает на компьютере сервера, а клиентские программы могут одновременно подключаться к ней через Интернет из любой точки мира. Если на сервере не предусмотрена обработка одновременных запросов от многих клиентов, один из клиентов может монополизировать доступ к серверной программе в течение длительного времени. Во избежание подобных ситуаций следует прибегнуть к помощи потоков исполнения.

Всякий раз, когда серверная программа устанавливает новое сокетное соединение, т.е. в результате вызова метода `accept()` возвращается сокет, запускается новый поток исполнения для подключения *данного* клиента к серверу. После этого происходит возврат в основную программу, которая переходит в режим ожидания следующего соединения. Для того чтобы все это произошло, в серверной программе следует организовать приведенный ниже основной цикл.

```
while (true)
{
  Socket incoming = s.accept();
  var r = new ThreadedEchoHandler(incoming);

  var t = new Thread(r);
  t.start();
}
```

Класс `ThreadedEchoHandler` реализует интерфейс `Runnable` и в своем методе `run()` поддерживает взаимодействие с клиентской программой:

```
class ThreadedEchoHandler implements Runnable
{
  . . .
  public void run()
  {
    try (InputStream inStream = incoming.getInputStream();
       OutputStream outStream = incoming.getOutputStream())
    {
      обработать полученный запрос и отправить ответ
    }
```

```
    catch(IOException e)
    {
      обработать исключение
    }
  }
}
```

Когда новый поток исполнения запускается при каждом соединении, несколько клиентских программ могут одновременно подключаться к серверу. Это нетрудно проверить, выполнив следующие действия.

1. Скомпилируйте и запустите на выполнение серверную программу, исходный код которой приведен в листинге 4.4.
2. Откройте несколько окон утилиты `telnet` (рис. 4.5).
3. Переходя из одного окна в другое, введите команды. В итоге каждое отдельное окно утилиты `telnet` будет взаимодействовать с серверной программой независимо от других окон.
4. Чтобы разорвать соединение и закрыть окно утилиты `telnet`, нажмите комбинацию клавиш <Ctrl+C>.

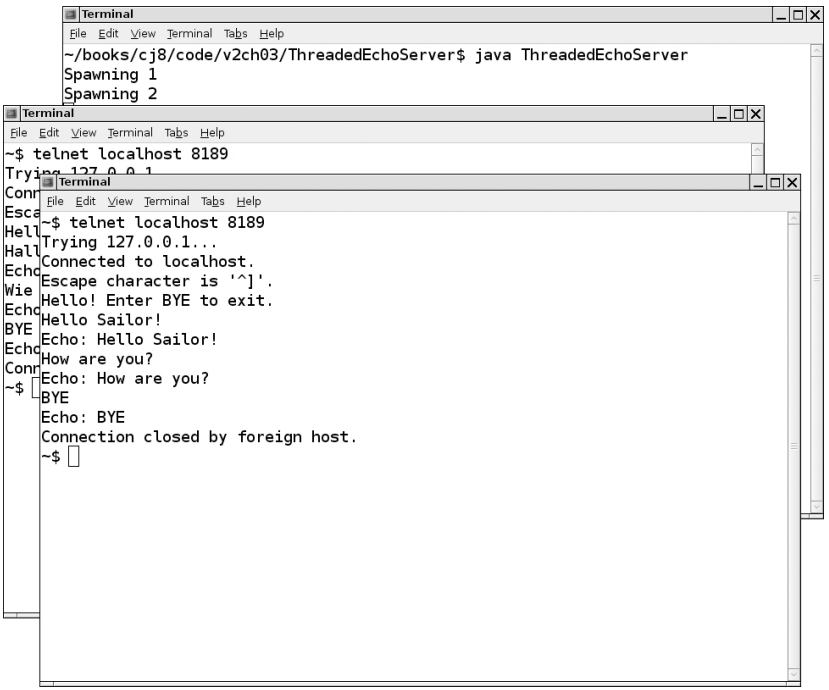


Рис. 4.5. Сеанс одновременной связи нескольких клиентов с сервером

НА ЗАМЕТКУ! В рассматриваемой здесь программе для каждого соединения порождается отдельный поток исполнения. Такой прием не вполне подходит для высокопроизводительного сервера. Более эффективной работы сервера можно добиться, используя средства из пакета **`java.nio`**. Дополнительные сведения по данному вопросу можно получить, обратившись по адресу **`https://www.ibm.com/developerworks/java/library/j-javaio/`**.

Листинг 4.4. Исходный код из файла **threaded/ThreadedEchoServer.java**

```
1  package threaded;
2
3  import java.io.*;
4  import java.net.*;
5  import java.nio.charset.*;
6  import java.util.*;
7
8  /**
9   * В этой программе реализуется многопоточный сервер,
10  * прослушивающий порт 8189 и передающий обратно все данные,
11  * полученные от всех клиентов
12  * @author Cay Horstmann
13  * @version 1.23 2018-03-17
14  */
15 public class ThreadedEchoServer
16 {
17   public static void main(String[] args )
18   {
19     try (var s = new ServerSocket(8189))
20     {
21       int i = 1;
22
23       while (true)
24       {
25         Socket incoming = s.accept();
26         System.out.println("Spawning " + i);
27         Runnable r = new ThreadedEchoHandler(incoming);
28         var t = new Thread(r);
29         t.start();
30         i++;
31       }
32     }
33     catch (IOException e)
34     {
35       e.printStackTrace();
36     }
37   }
38 }
39
40 /**
41  * Этот класс обрабатывает данные, получаемые сервером
42  * от клиента через одно сокетное соединение
43  */
44 class ThreadedEchoHandler implements Runnable
45 {
46   private Socket incoming;
47
48   /**
49      Конструирует обработчик
50      @param incomingSocket Входящий сокет
51   */
52   public ThreadedEchoHandler(Socket incomingSocket)
53   {
```

```
54      incoming = incomingSocket;
55    }
56
57    public void run()
58    {
59      try (InputStream inStream =
60           incoming.getInputStream();
61           OutputStream outStream =
62                       incoming.getOutputStream();
63           var in = new Scanner(inStream,
64                                StandardCharsets.UTF_8);
65           var out = new PrintWriter(
66                     new OutputStreamWriter(outStream,
67                         StandardCharsets.UTF_8),
68                         true /* autoFlush */))
69      {
70        out.println("Hello! Enter BYE to exit.");
71
72        // передать обратно данные, полученные от клиента
73        var done = false;
74        while (!done && in.hasNextLine())
75        {
76          String line = in.nextLine();
77          out.println("Echo: " + line);
78          if (line.trim().equals("BYE"))
79            done = true;
80        }
81      }
82      catch (IOException e)
83      {
84        e.printStackTrace();
85      }
86    }
87  }
```

4.2.3. Полузакрытие

Полузакрытие обеспечивает возможность прервать передачу данных на одной стороне сокетного соединения, продолжая в то же время прием данных от другой стороны. Рассмотрим типичную ситуацию. Допустим, данные направляются на сервер, но заранее неизвестно, какой именно объем данных требуется передать. Если речь идет о файле, то его закрытие, по существу, означает завершение передачи данных. Если же закрыть сокет, то соединение с сервером будет немедленно разорвано.

Для преодоления подобного затруднения служит полузакрытие. Если закрыть поток вывода через сокет, то для сервера это будет означать завершение передачи данных запроса. При этом поток ввода остается открытым, позволяя получить ответ от сервера. Код, реализующий механизм полузакрытия на стороне клиента, приведен ниже.

```
try (var socket = new Socket(host, port))
{
  var in = new Scanner(socket.getInputStream(), "UTF-8");
```

```
  var writer = new PrintWriter(socket.getOutputStream());
  // передать данные запроса
  writer.print(. . .);
  writer.flush();
  socket.shutdownOutput();
  // теперь сокет полузакрыт
  // принять данные ответа
  while (in.hasNextLine() != null)
  {
    String line = in.nextLine();
    . . .
  }
}
```

Серверная программа просто читает данные из потока ввода до тех пор, пока не закроется поток вывода на другом конце соединения. Очевидно, что такой подход применим только для служб однократного действия по сетевым протоколам, подобным HTTP, где клиент устанавливает соединение с сервером, передает запрос, получает ответ, после чего соединение разрывается.

java.net.Socket 1.0

- **void shutdownOutput() 1.3**
  Устанавливает поток вывода в состояние завершения.
- **void shutdownInput() 1.3**
  Устанавливает поток ввода в состояние завершения.
- **boolean isOutputShutdown() 1.4**
  Возвращает логическое значение **true**, если вывод данных был остановлен.
- **boolean isInputShutdown() 1.4**
  Возвращает логическое значение **true**, если ввод данных был остановлен.

4.2.4. Прерываемые сокеты

При подключении через сокет текущий поток исполнения блокируется до тех пор, пока соединение не будет установлено, или же до истечения времени ожидания. Аналогично, если пытаться принять данные через сокет, текущий поток приостановит свое исполнение до успешного завершения операции или до истечения времени ожидания. (Для передачи данных время ожидания не устанавливается.)

В прикладных программах, работающих в диалоговом режиме, пользователям желательно предоставить возможность прервать слишком затянувшийся процесс установления соединения через сокет. Но если поток исполнения блокирован для нереагирующего сокета, то разблокировать его не удастся, вызвав метод `interrupt()`.

Для прерывания сокетных операций служит класс `SocketChannel`, предоставляемый в пакете `java.nio`. Объект типа `SocketChannel` создается следующим образом:

```
SocketChannel channel = SocketChannel.open(
            new InetSocketAddress(host, port));
```

У канала отсутствуют связанные с ним потоки ввода-вывода. Вместо этого в канале предоставляются методы `read()` и `write()`, использующие объекты типа `Buffer`. (Подробнее о буферах из системы ввода-вывода NIO см. в главе 2.) Эти методы объявляются в интерфейсах `ReadableByteChannel` и `WritableByteChannel`. Если же нет желания иметь дело с буферами, для чтения из канала типа `SocketChannel` можно воспользоваться объектом типа `Scanner`. Для этой цели в классе `Scanner` предусмотрен следующий конструктор с параметром типа `ReadableByteChannel`:

```
var in = new Scanner(channel, StandardCharsets.UTF_8);
```

Чтобы превратить канал в поток вывода, применяется статический метод `Channels.newOutputStream()`:

```
OutputStream outStream = Channels.newOutputStream(channel);
```

Вот, собственно, и все, что нужно сделать для прерывания сокетной операции. Если же поток исполнения будет прерван в процессе установления соединения, чтения или записи, соответствующая операция завершится генерированием исключения.

В примере программы, исходный код которой приведен в листинге 4.5, демонстрируется применение прерываемых и блокирующих сокетов. Сервер передает числовые данные, имитируя прерывание их передачи после десятого числа. Если щелкнуть на любой кнопке, запустится поток исполнения, устанавливающий соединение с сервером и выводящий на экран передаваемые данные. В первом потоке исполнения используется прерываемый сокет, а во втором — блокирующий. Если щелкнуть на кнопке Cancel (Отмена) во время вывода первых десяти чисел, то прервется исполнение любого из двух потоков.

Если щелкнуть на кнопке Cancel после передачи первых десяти чисел, то прервется исполнение только первого потока. Блокировка второго потока исполнения будет продолжаться до тех пор, пока сервер не разорвет окончательно соединение (рис. 4.6).

Рис. 4.6. Прерывание сокета

Листинг 4.5. Исходный код из файла **interruptible/InterruptibleSocketTest.java**

```
1   package interruptible;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.util.*;
6   import java.net.*;
7   import java.io.*;
8   import java.nio.charset.*;
9   import java.nio.channels.*;
10  import javax.swing.*;
11
12  /**
13   * В этой программе демонстрируется прерывание
14   * сокета через канал
15   * @author Cay Horstmann
16   * @version 1.05 2018-03-17
17   */
18  public class InterruptibleSocketTest
19  {
20    public static void main(String[] args)
21    {
22      EventQueue.invokeLater(() ->
23       {
24         var frame = new InterruptibleSocketFrame();
25         frame.setTitle("InterruptibleSocketTest");
26         frame.setDefaultCloseOperation(
27               JFrame.EXIT_ON_CLOSE);
28         frame.setVisible(true);
29       });
30    }
31  }
32
33  class InterruptibleSocketFrame extends JFrame
34  {
35    private Scanner in;
36    private JButton interruptibleButton;
37    private JButton blockingButton;
38    private JButton cancelButton;
39    private JTextArea messages;
40    private TestServer server;
41    private Thread connectThread;
42
43    public InterruptibleSocketFrame()
44    {
45      var northPanel = new JPanel();
46      add(northPanel, BorderLayout.NORTH);
47
48      final int TEXT_ROWS = 20;
49      final int TEXT_COLUMNS = 60;
50      messages = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
51      add(new JScrollPane(messages));
52
53      interruptibleButton = new JButton("Interruptible");
54      blockingButton = new JButton("Blocking");
```

```
55
56      northPanel.add(interruptibleButton);
57      northPanel.add(blockingButton);
58
59      interruptibleButton.addActionListener(event ->
60        {
61          interruptibleButton.setEnabled(false);
62          blockingButton.setEnabled(false);
63          cancelButton.setEnabled(true);
64          connectThread = new Thread(() ->
65            {
66              try
67              {
68                connectInterruptibly();
69              }
70              catch (IOException e)
71              {
72                messages.append(
73                         "\nInterruptibleSocketTest."
74                       + "connectInterruptibly: " + e);
75              }
76            });
77          connectThread.start();
78        });
79
80      blockingButton.addActionListener(event ->
81       {
82         interruptibleButton.setEnabled(false);
83         blockingButton.setEnabled(false);
84         cancelButton.setEnabled(true);
85         connectThread = new Thread(() ->
86           {
87             try
88             {
89               connectBlocking();
90             }
91             catch (IOException e)
92             {
93               messages.append(
94                        "\nInterruptibleSocketTest."
95                      + "connectBlocking: " + e);
96             }
97           });
98         connectThread.start();
99       });
100
101     cancelButton = new JButton("Cancel");
102     cancelButton.setEnabled(false);
103     northPanel.add(cancelButton);
104     cancelButton.addActionListener(event ->
105       {
106         connectThread.interrupt();
107         cancelButton.setEnabled(false);
108       });
109     server = new TestServer();
110     new Thread(server).start();
```

```
111     pack();
112   }
113
114   /**
115    * Соединяет с проверяемым сервером,
116    * используя прерываемый ввод-вывод
117    */
118   public void connectInterruptibly() throws IOException
119   {
120     messages.append("Interruptible:\n");
121     try (SocketChannel channel = SocketChannel
122        .open(new InetSocketAddress("localhost", 8189)))
123     {
124       in = new Scanner(channel, StandardCharsets.UTF_8);
125       while (!Thread.currentThread().isInterrupted())
126       {
127         messages.append("Reading ");
128         if (in.hasNextLine())
129         {
130           String line = in.nextLine();
131           messages.append(line);
132           messages.append("\n");
133         }
134       }
135     }
136     finally
137     {
138       EventQueue.invokeLater(() ->
139         {
140           messages.append("Channel closed\n");
141           interruptibleButton.setEnabled(true);
142           blockingButton.setEnabled(true);
143         });
144     }
145   }
146
147   /**
148    * Соединяет с проверяемым сервером,
149    * используя блокирующий ввод-вывод
150    */
151   public void connectBlocking() throws IOException
152   {
153     messages.append("Blocking:\n");
154     try (var sock = new Socket("localhost", 8189))
155     {
156       in = new Scanner(sock.getInputStream(),
157                        StandardCharsets.UTF_8);
158       while (!Thread.currentThread().isInterrupted())
159       {
160         messages.append("Reading ");
161         if (in.hasNextLine())
162         {
163           String line = in.nextLine();
164           messages.append(line);
165           messages.append("\n");
166         }
```

```
167       }
168     }
169     finally
170     {
171       EventQueue.invokeLater(() ->
172        {
173          messages.append("Socket closed\n");
174          interruptibleButton.setEnabled(true);
175          blockingButton.setEnabled(true);
176        });
177     }
178   }
179
180   /**
181    * Многопоточный сервер, прослушивающий порт 8189 и
182    * посылающий клиентам числа, имитируя зависание
183    * после передачи 10 чисел
184    *
185    */
186   class TestServer implements Runnable
187   {
188     public void run()
189     {
190       try (var s = new ServerSocket(8189))
191       {
192         while (true)
193         {
194           Socket incoming = s.accept();
195           Runnable r = new TestServerHandler(incoming);
196           new Thread(r).start();
197         }
198       }
199       catch (IOException e)
200       {
201         messages.append("\nTestServer.run: " + e);
202       }
203     }
204   }
205
206   /**
207    * Этот класс обрабатывает данные, получаемые
208    * сервером от клиента через одно сокетное соединение
209    */
210   class TestServerHandler implements Runnable
211   {
212     private Socket incoming;
213     private int counter;
214
215     /**
216      * Конструирует обработчик
217      * @param i Входящий сокет
218      */
219     public TestServerHandler(Socket i)
220     {
221       incoming = i;
222     }
```

```
223
224     public void run()
225     {
226       try
227       {
228         try
229         {
230           OutputStream outStream =
231                 incoming.getOutputStream();
232           var out = new PrintWriter(
233           new OutputStreamWriter(outStream,
234                     StandardCharsets.UTF_8),
235           true /* автоматическая очистка */);
236           while (counter < 100)
237           {
238             counter++;
239             if (counter <= 10) out.println(counter);
240             Thread.sleep(100);
241           }
242         }
243         finally
244         {
245           incoming.close();
246           messages.append("Closing server\n");
247         }
248       }
249       catch (Exception e)
250       {
251         messages.append("\nTestServerHandler.run: " + e);
252       }
253     }
254   }
255 }
```

java.net.InetSocketAddress 1.4

- **InetSocketAddress(String *hostname*, int *port*)**

  Создает объект адреса с указанными именем хоста (т.е. сетевого узла) и номером порта, преобразуя имя узла в адрес при установлении соединения. Если преобразовать имя хоста в адрес не удается, устанавливается логическое значение **true** свойства **unresolved**.
- **boolean isUnresolved()**

  Возвращает логическое значение **true**, если для данного объекта не удается преобразовать имя хоста в адрес.

java.nio.channels.SocketChannel 1.4

- **static SocketChannel open(SocketAddress *address*)**

  Открывает канал для сокета и связывает его с удаленным хостом по указанному адресу.

java.nio.channels.Channels 1.4

- **static InputStream newInputStream(ReadableByteChannel *channel*)**
  Создает поток ввода для чтения данных из указанного канала.
- **static OutputStream newOutputStream(WritableByteChannel *channel*)**
  Создает поток вывода для записи данных в указанный канал.

4.3. Получение данных из Интернета

Чтобы получить доступ к веб-серверам из программы на Java, требуется более высокий уровень сетевого взаимодействия, чем установление соединения через сокет и выдача HTTP-запросов. В последующих разделах будут рассмотрены классы, предоставляемые для этой цели в библиотеке Java.

4.3.1. URL и URI

Классы `URL` и `URLConnection` инкапсулируют большую часть внутреннего механизма извлечения данных с удаленного веб-сайта. Объект типа `URL` создается следующим образом:

```
URL url = new URL(символьная строка с URL);
```

Если требуется только извлечь содержимое из указанного ресурса, достаточно вызвать метод `openStream()` из класса `URL`. Этот метод возвращает объект типа `InputStream`. Поток ввода данного типа можно использовать обычным образом, например, создать объект типа `Scanner`:

```
InputStream inStream = url.openStream();
var in = new Scanner(inStream, StandardCharsets.UTF_8);
```

В пакете `java.net` отчетливо различаются унифицированные *указатели* ресурсов (URL) и унифицированные *идентификаторы* ресурсов (URI). В частности, URI — это лишь синтаксическая конструкция, содержащая различные части символьной строки, обозначающей веб-ресурс. URL — это особая разновидность идентификатора URI с исчерпывающими данными о местоположении ресурса. Имеются и такие URI, как, например, `mailto:cay@hortsmann.com`, которые не являются указателями ресурсов, потому что по ним нельзя обнаружить какие-нибудь данные. Такой URI называется унифицированным *именем* ресурса (URN).

В классе `URI` из библиотеки Java отсутствуют методы доступа к ресурсу по указанному идентификатору, поскольку этот класс предназначен только для синтаксического анализа символьной строки, обозначающей ресурс. В отличие от него, класс `URL` позволяет открыть поток ввода-вывода для данного ресурса. Поэтому в классе `URL` допускается взаимодействие только по тем протоколам и схемам, которые поддерживаются в библиотеке Java, в том числе `http:`, `https:` и `ftp:` — для Интернета, `file:` — для локальной файловой системы, а также `jar:` — для обращения к архивным JAR-файлам.

Синтаксический анализ URI — непростая задача, поскольку идентификаторы ресурсов могут иметь сложную структуру. В качестве примера ниже приведены URI с замысловатой структурой.

```
http:/google.com?q=Beach+Chalet
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

В обозначении идентификаторов URI задаются правила их построения. Структура URI выглядит следующим образом:

```
[схема:]специальная_часть_схемы[#фрагмент]
```

где квадратные скобки обозначают необязательную часть, а двоеточие и знак # служат в качестве разделителей. Если *схема*: присутствует как составная часть в идентификаторе URI, то он называется *абсолютным*, а иначе — *относительным*. Абсолютный URI называется *непрозрачным*, если *специальная_часть_схемы* не начинается с косой черты (/), как, например, показано ниже.

```
mailto:cay@horstmann.com
```

Все абсолютные, непрозрачные URI и все относительные URL имеют *иерархическую структуру*. Например:

```
http://horstmann.com/index.html
../../java/net/Socket.html#Socket()
```

Составляющая *специальная_часть_схемы* иерархического URI имеет следующую структуру:

```
[//полномочия][путь][?запрос]
```

И здесь квадратные скобки обозначают необязательную часть. Составляющая *полномочия* в URI серверов имеет приведенную ниже форму, где элемент *порт* должен иметь целочисленное значение.

```
[сведения_о_пользователе@]хост[:порт]
```

В документе RFC 2396, стандартизирующем идентификаторы URI, допускается также механизм указания составляющей *полномочия* в другом формате на основе данных из реестра. Но он не получил широкого распространения.

Одно из назначений класса `URI` состоит в синтаксическом анализе отдельных составляющих идентификатора. Они извлекаются с помощью перечисленных ниже методов.

```
GetScheme()
getSchemeSpecificPart()
getAuthority()
getUserInfo()
getHost()
getPort()
getPath()
getQuery()
getFragment()
```

Другое назначение класса `URI` состоит в обработке абсолютных и относительных идентификаторов. Так, если имеются абсолютный и относительный идентификаторы URI:

```
http://docs.mycompany.com/api/java/net/ServerSocket.html
```

и

```
../../java/net/Socket.html#Socket()
```

их можно объединить в абсолютный URI следующим образом:

```
http://docs.mycompany.com/api/java/net/Socket.html#Socket()
```

Такой процесс называется *преобразованием адресов* относительного URI. Обратный процесс называется *преобразованием абсолютных адресов в относительные*. Например, имея *базовый* URI:

```
http://docs.mycompany.com/api
```

можно преобразовать следующий абсолютный URI:

```
http://docs.company.com/api/java/lang/String.html
```

в приведенный ниже относительный URI.

```
java/lang/String.html
```

Для выполнения обоих видов преобразования в классе `URI` предусмотрены два соответствующих метода:

```
relative = base.relativize(combined);
combined = base.resolve(relative);
```

4.3.2. Извлечение данных средствами класса `URLConnection`

Для получения дополнительных сведений о веб-ресурсе следует воспользоваться классом `URLConnection`, предоставляющим намного больше средств управления доступом к веб-ресурсам, чем более простой класс `URL`. Для работы с объектом типа `URLConnection` необходимо тщательно спланировать и выполнить следующие действия.

1. Вызвать метод `openConnection()` из класса `URL` для получения объекта типа `URLConnection` следующим образом:

   ```
   URLConnection connection = url.openConnection();
   ```

2. Задать свойства запроса с помощью перечисленных ниже методов.

   ```
   SetDoInput()
   setDoOutput()
   setIfModifiedSince()
   setUseCaches()
   setAllowUserInteraction()
   setRequestProperty()
   setConnectTimeout()
   setReadTimeout()
   ```

3. Эти методы будут подробно рассматриваться далее.

4. Установить соединение с удаленным ресурсом с помощью метода `connect()`:

   ```
   connection.connect();
   ```

5. Помимо создания сокета, для установления соединения с веб-сервером этот метод запрашивает также у сервера *данные заголовка*.

6. После подключения к веб-серверу становятся доступными поля заголовка. Обращаться к ним можно с помощью универсальных методов `getHeaderFieldKey()` и `getHeaderField()`. Кроме того, для удобства разработки предусмотрены перечисленные ниже методы обработки стандартных полей запроса.

```
getContentType()
getContentLength()
getContentEncoding()
getDate()
getExpiration()
getLastModified()
```

7. Наконец, для доступа к данным указанного ресурса следует вызвать метод `getInputStream()`, предоставляющий поток ввода для чтения данных. (Это тот же поток ввода, который возвращается методом `openStream()` из класса `URL`.) Существует также метод `getContent()`, но он не такой удобный. Для обработки содержимого стандартных типов, например текста (`text/plain`) или изображений (`image/gif`), придется воспользоваться классами из пакета `com.sun`. Кроме того, можно зарегистрировать собственные обработчики содержимого, но они в данной книге не рассматриваются.

НА ЗАМЕТКУ! Некоторые разработчики, пользующиеся классом **`URLConnection`**, ошибочно считают, что методы **`getInputStream()`** и **`getOutpuStream()`** аналогичны одноименным методам из класса **`Socket`**. Это не совсем так. Класс **`URLConnection`** способен выполнять много других функций, в том числе обрабатывать заголовки запросов и ответов. Поэтому рекомендуется строго придерживаться указанной выше последовательности действий.

Рассмотрим методы из класса `URLConnection` более подробно. В нем имеется ряд методов, задающих свойства соединения еще до подключения к веб-серверу. Наиболее важными среди них являются методы `setDoInput()` и `setDoOutput()`. По умолчанию при соединении предоставляется поток ввода для приема данных с веб-сервера, но не поток вывода для передачи данных. Чтобы получить поток вывода (например, с целью разместить данные на веб-сервере), необходимо сделать следующий вызов:

```
connection.setDoOutput(true);
```

Далее можно установить ряд заголовков запроса и послать их веб-серверу в составе единого запроса. Ниже приведен пример заголовков запроса.

```
GET www.server.com/index.html HTTP/1.0
Referer: http://www.somewhere.com/links.html
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.4)
Host: www.server.com
Accept: text/html, image/gif, image/jpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: orangemilano=192218887821987
```

Метод `setIfModifiedSince()` служит для уведомления о том, что требуется получить только те данные, которые были изменены после определенной даты.

Наконец, с помощью метода `setRequestProperty()` можно установить пару "имя–значение", имеющую определенный смысл для конкретного протокола. Формат заголовка запроса по сетевому протоколу HTTP описан в документе RFC 2616. Некоторые его параметры не очень хорошо документированы, поэтому за дополнительными разъяснениями зачастую приходится обращаться к опыту других программистов. Так, для доступа к защищенной паролем веб-странице необходимо выполнить следующие действия.

1. Составить символьную строку из имени пользователя, двоеточия и пароля:

```
String input = username + ":" + password;
```

2. Перекодировать полученную в итоге символьную строку по алгоритму кодирования Base64, как показано ниже. (Этот алгоритм преобразует последовательность байтов в последовательность символов в коде ASCII.)

```
Base64.Encoder encoder = Base64.getEncoder();
String encoding = encoder.encodeToString(
                  input.getBytes(StandardCharsets.UTF_8));
```

3. Вызвать метод `setRequestProperty()` с именем свойства `"Authorization"` и значением `"Basic " + encoding`, как показано ниже.

```
connection.setRequestProperty("Authorization",
                              "Basic " + encoding);
```

СОВЕТ. Здесь рассматривается способ обращения к защищенной паролем веб-странице. Для доступа к защищенному паролем FTP-файлу применяется совершенно другой подход. В этом случае достаточно сформировать URL следующего вида:

ftp://*имя_пользователя*:*пароль*@ftp.*ваш_сервер*.com/pub/file.txt

После вызова метода `connect()` можно запросить данные заголовка из ответа. Рассмотрим сначала способ перечисления всех полей заголовка. Создатели рассматриваемого здесь класса посчитали нужным создать собственный способ перебора полей. Так, в результате вызова приведенного ниже метода получается *n*-й ключ заголовка, причем нумерация начинается с единицы! В итоге возвращается пустое значение `null`, если *n* равно нулю или больше общего количества полей заголовка.

```
String key = connection.getHeaderFieldKey(n);
```

Но для определения количества полей не предусмотрено никакого другого метода. Чтобы перебрать все поля, приходится вызывать метод `getHeaderFieldKey()` до тех пор, пока не будет получено пустое значение `null`. Аналогично при вызове следующего метода возвращается значение из *n*-го поля:

```
String value = connection.getHeaderField(n);
```

Метод `getHeaderFields()` возвращает объект типа `Map` с полями заголовка:

```
Map<String,List<String>> headerFields =
                         connection.getHeaderFields();
```

В качестве примера ниже приведен ряд полей заголовка из типичного ответа на запрос по сетевому протоколу HTTP.

```
Date: Wed, 27 Aug 2008 00:15:48 GMT
Server: Apache/2.2.2 (Unix)
Last-Modified: Sun, 22 Jun 2008 20:53:38 GMT
Accept-Ranges: bytes
Content-Length: 4813
Connection: close
Content-Type: text/html
```

НА ЗАМЕТКУ! Получить в ответ строку состояния (например, **"HTTP/1.1 200 OK"**) можно, сделав вызов **connection.getHeaderField(0)** или **headerFields.get(null)**.

Для удобства разработки предусмотрены шесть методов, получающих значения из наиболее употребительных полей заголовка и приводящие эти значения к соответствующим числовым типам по мере необходимости. Все эти удобные методы перечислены в табл. 4.1. В методах, возвращающих значения типа `long`, отсчет количества возвращаемых секунд начинается с полуночи 1 января 1970 г.

Таблица 4.1. Удобные методы, получающие значения полей заголовка из ответа на запрос

Имя поля (ключа)	Имя метода	Возвращаемое значение
Date	**getDate**	**long**
Expires	**getExpiration**	**long**
Last-Modified	**getLastModified**	**long**
Content-Length	**getContentLength**	**int**
Content-Type	**getContentType**	**String**
Content-Encoding	**getContentEncoding**	**String**

В примере программы из листинга 4.6 предоставляется возможность поэкспериментировать с соединениями по URL. Запустив программу, вы можете указать в командной строке конкретный URL, имя пользователя и пароль:

```
java urlConnection.URLConnectionTest http://www.ваш_сервер.com
     пользователь пароль
```

В итоге программа выведет на экран следующее.

- Все ключи и значения из полей заголовка.
- Значения, возвращаемые шестью служебными методами доступа к наиболее употребительным полям заголовка (см. табл. 4.1).
- Первые 10 символьных строк из запрашиваемого ресурса.

Листинг 4.6. Исходный код из файла **urlConnection/URLConnectionTest.java**

```
1  package urlConnection;
2
3  import java.io.*;
4  import java.net.*;
5  import java.nio.charset.*;
6  import java.util.*;
7
8  /**
9   * В этой программе устанавливается соединение по
```

```
 * заданному URL и отображаются данные заголовка из
 * получаемого ответа, а также первые 10 строк
 * запрашиваемых данных. Для этого в командной строке
 * следует указать конкретный URL, дополнительно имя
 * пользователя и пароль (для элементарной аутентификации
 * по сетевому протоколу HTTP)
 * @version 1.12 2018-03-17
 * @author Cay Horstmann
 */
public class URLConnectionTest
{
  public static void main(String[] args)
  {
    try
    {
      String urlName;
      if (args.length > 0) urlName = args[0];
      else urlName = "http://horstmann.com";

      var url = new URL(urlName);
      URLConnection connection = url.openConnection();

      // установить имя пользователя и пароль, если они
      // указаны в командной строке

      if (args.length > 2)
      {
        String username = args[1];
        String password = args[2];
        String input = username + ":" + password;
        Base64.Encoder encoder = Base64.getEncoder();
        String encoding = encoder.encodeToString(
               input.getBytes(StandardCharsets.UTF_8));
        connection.setRequestProperty("Authorization",
                   "Basic " + encoding);
      }

      connection.connect();

      // вывести поля заголовка

      Map<String, List<String>> headers =
                  connection.getHeaderFields();
      for (Map.Entry<String, List<String>> entry :
           headers.entrySet())
      {
        String key = entry.getKey();
        for (String value : entry.getValue())
          System.out.println(key + ": " + value);
      }

      // вывести значения полей заголовка,
      // используя удобные методы

      System.out.println("----------");
      System.out.println("getContentType: "
                         + connection.getContentType());
```

```
67       System.out.println("getContentLength: "
68                          + connection.getContentLength());
69       System.out.println("getContentEncoding: "
70                         + connection.getContentEncoding());
71       System.out.println("getDate: "
72                          + connection.getDate());
73       System.out.println("getExpiration: "
74                          + connection.getExpiration());
75       System.out.println("getLastModifed: "
76                          + connection.getLastModified());
77       System.out.println("----------");
78
79       String encoding = connection.getContentEncoding();
80       if (encoding == null) encoding = "UTF-8";
81       try (var in = new Scanner(
82            connection.getInputStream(), encoding))
83       {
84         // вывести первые десять строк
85         // запрашиваемого содержимого
86
87         for (int n = 1; in.hasNextLine() && n <= 10; n++)
88           System.out.println(in.nextLine());
89         if (in.hasNextLine()) System.out.println(". . .");
90       }
91     }
92     catch (IOException e)
93     {
94       e.printStackTrace();
95     }
96   }
97 }
```

java.net.URL 1.0

- **InputStream openStream()**

  Открывает поток ввода для чтения данных из ресурса.
- **URLConnection openConnection()**

  Возвращает объект типа **URLConnection**, управляющий соединением с ресурсом.

java.net.URLConnection 1.0

- **void setDoInput(boolean *doInput*)**
- **boolean getDoInput()**

  Если задано логическое значение **true** параметра ***doInput***, пользователь может принимать вводимые данные из текущего объекта типа **URLConnection**.
- **void setDoOutput(boolean *doOutput*)**
- **boolean getDoOutput()**

  Если задано логическое значение **true** параметра ***doOutput***, пользователь может передавать выводимые данные в текущий объект типа **URLConnection**.

java.net.URLConnection 1.0 *(продолжение)*

- **void setIfModifiedSince(long *time*)**
- **long getIfModifiedSince()**

  Свойство **ifModifiedSince** настраивает данный объект типа **URLConnection** на извлечение только тех данных, которые были изменены после указанного момента времени. Время задается в секундах, начиная с полуночи 1 января 1970 г. по Гринвичу.
- **void setConnectTimeout(int *timeout*)** 5.0
- **int getConnectTimeout()** 5.0

  Устанавливают или возвращают величину времени ожидания (в миллисекундах) для соединения. Если время ожидания истечет до установления соединения, метод **connect()** из соответствующего потока ввода сгенерирует исключение типа **SocketTimeoutException**.
- **void setReadTimeout(int *timeout*)** 5.0
- **int getReadTimeout()** 5.0

  Устанавливают или возвращают величину времени ожидания (в миллисекундах) для чтения данных. Если время ожидания истечет до успешного завершения операции чтения, метод **read()** сгенерирует исключение типа **SocketTimeoutException**.
- **void setRequestProperty(String *key*, String *value*)**

  Устанавливает значение в поле заголовка.
- **Map<String, List<String>> getRequestProperties()** 1.4

  Возвращает отображение со свойствами запроса. Все свойства по одному и тому же ключу вносятся в список.
- **void connect()**

  Устанавливает соединение с удаленным ресурсом и получает данные заголовка из ответа.
- **Map<String,List<String>> getHeaderFields()** 1.4

  Возвращает отображение с полями заголовка из ответа. Все свойства одного и того же ключа вносятся в список.
- **String getHeaderFieldKey(int *n*)**

  Возвращает ключ ***n***-го поля заголовка из ответа или пустое значение **null**, если ***n*** меньше или равно нулю или превышает количество полей.
- **String getHeaderField(int *n*)**

  Возвращает значение ***n***-го поля заголовка из ответа или пустое значение **null**, если ***n*** меньше или равно нулю или превышает количество полей.
- **int getContentLength()**

  Возвращает длину доступного содержимого или **-1**, если длина неизвестна.
- **String getContentType()**

  Возвращает тип содержимого, например, **text/plain** или **image/gif**.
- **String getContentEncoding()**

  Возвращает кодировку содержимого, например **gzip**. Применяется редко, потому что используемая по умолчанию кодировка не всегда указывается в поле **identity** заголовка **Content-Encoding**.
- **long getDate()**
- **long getExpiration()**
- **long getLastModifed()**

  Возвращают время создания, последней модификации ресурса или время, когда истекает срок действия ресурса. Время указывается в секундах, начиная с 1 января 1970 г. по Гринвичу.

java.net.URLConnection 1.0 *(окончание)*

- **InputStream getInputStream()**
- **OutputStream getOutputStream()**

  Возвращают поток ввода для чтения данных из ресурса или вывода для записи данных в ресурс.
- **Object getContent()**

  Выбирает подходящий обработчик содержимого для чтения данных из ресурса. Этот метод вряд ли полезен для чтения данных стандартного типа, например, **text/plain** или **image/gif**, кроме тех случаев, когда требуется создать собственный обработчик этих типов данных.

4.3.3. Отправка данных формы

В предыдущем разделе описывался способ приема данных с веб-сервера, в этом разделе рассматривается способ передачи данных из клиентской программы на веб-сервер, а также другим программам, которые может вызывать веб-сервер. Для передачи данных из браузера на веб-сервер нужно заполнить форму, аналогичную приведенной на рис. 4.7.

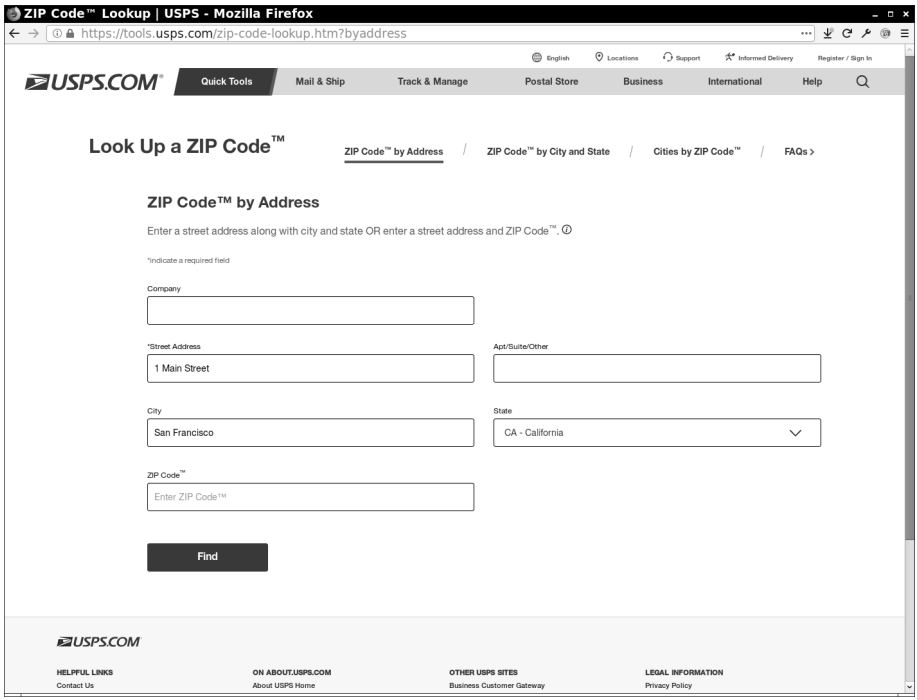


Рис. 4.7. HTML-форма

Когда пользователь щелкает на кнопке Submit (Отправить), данные, введенные в текстовых полях, а также сведения о состоянии флажков и кнопок-переключателей передаются на веб-сервер. Получив данные, введенные пользователем в форме, веб-сервер вызывает программу для их последующей обработки.

Существует целый ряд технологий, позволяющих веб-серверу вызывать программы для обработки данных. Наиболее часто для этой цели используются сервлеты на Java, платформы JavaServer Faces и Microsoft ASP (Active Server Pages — активные серверные страницы), а также сценарии CGI (Common Gateway Interface — общий шлюзовой интерфейс).

Программа, выполняющаяся на стороне сервера, обрабатывает данные, введенные пользователем в форме, и формирует новую HTML-страницу, которую веб-сервер передает обратно браузеру. Последовательность действий по обработке данных из формы схематически показана на рис. 4.8. Ответная страница, сформированная сервером, может содержать новые данные (например, результаты поиска) или только подтверждение о получении введенных данных. Здесь и далее не рассматриваются вопросы реализации серверных программ, а основное внимание уделяется написанию клиентских программ, предназначенных для взаимодействия с готовыми сценариями.

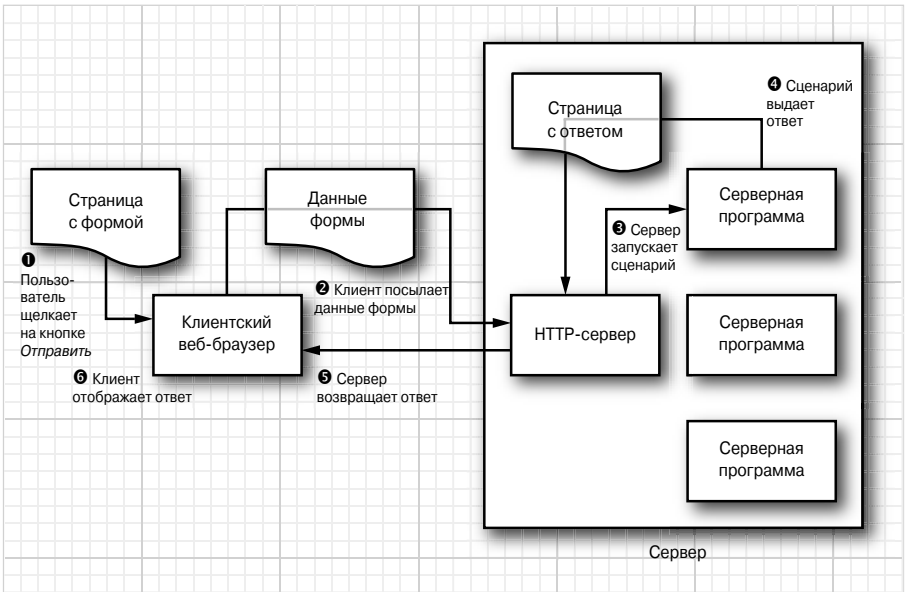


Рис. 4.8. Порядок обработки данных в серверной программе

При передаче данных на веб-сервер не имеет никакого значения, будет ли использован для их интерпретации сценарий CGI, сервлет или программа другого типа. Клиент посылает данные на веб-сервер в стандартном формате, а веб-сервер должен сам найти ту программу, которая выдаст нужный ответ.

Передача данных на веб-сервер может осуществляться по командам `GET` и `POST`. При выдаче команды `GET` параметры запроса указываются в конце URL в следующем формате:

`http://`*`хост`*`/`*`путь`*`?`*`запрос`*

Каждый параметр имеет вид *`имя=значение`*. Параметры разделяются знаками `&`. Значения параметров кодируются по схеме кодирования URL, которая подчиняется следующим правилам.

- Символы от A до Z, от a до z, от 0 до 9, а также знаки ., -, ~ и _ остаются без изменения.
- Все пробелы заменяются знаками +.
- Все остальные символы кодируются в кодировке UTF-8, а каждый байт преобразуется в вид %UV, где UV — двухзначное шестнадцатеричное число.

Например, название города и штата *San Francisco, CA* передается в закодированном виде как San+Francisco%2c+CA. Здесь шестнадцатеричное число 2c (или десятичное 44) обозначает запятую в кодировке UTF-8. Благодаря такому способу кодирования промежуточные программы не будут путаться в пробелах и смогут правильно интерпретировать другие символы.

На момент написания данной книги веб-сайт Google Maps (www.google.com/maps) принимал параметры запроса с именами q и hl, значения которых определяют местоположение и естественный язык в ответе. Чтобы получить карту местности по адресу Маркет-стрит, 1, г. Сан-Франциско на немецком языке, необходимо указать следующий URL:

```
http://www.google.com/maps?q=1+Market+Street+San+Francisco&hl=de
```

Очень длинные строки запроса могут выглядеть непривлекательно в большинстве браузеров, а в старых браузерах и промежуточных серверах накладывается ограничение на количество символов, включаемых в запрос по команде GET. Именно поэтому запрос по команде POST чаще всего употребляется для форм, содержащих немало данных. Параметры запроса по команде POST не следует включать в состав URL. Вместо этого следует получить поток вывода из объекта типа URLConnection и записать в него пары "имя–значение". Кроме того, значения, включаемые в URL, необходимо закодировать, разделив их знаком &.

Рассмотрим этот процесс более подробно. Для передачи данных серверной программе сначала создается объект типа URLConnection:

```
var url = new URL("http://хост/путь");
URLConnection connection = url.openConnection();
```

Затем вызывается метод setDoOutput(), чтобы установить соединение для передачи данных:

```
connection.setDoOutput(true);
```

Далее вызывается метод getOutputStream(), чтобы получить поток вывода. Для передачи текстовых данных поток вывода удобно инкапсулировать в объект типа PrintWriter следующим образом:

```
var out = new PrintWriter(connection.getOutputStream(),
                          StandardCharsets.UTF_8);
```

Теперь можно передать данные на сервер, как показано ниже.

```
out.print(name1 + "=" + URLEncoder.encode(value1, "UTF-8") + "&");
out.print(name2 + "=" + URLEncoder.encode(value2, "UTF-8"));
```

После передачи данных поток вывода закрывается следующим образом:

```
out.close();
```

Наконец, вызывается метод getInputStream(), чтобы прочитать ответ с сервера.

Рассмотрим конкретный практический пример. Веб-сайт, доступный по адресу `https://tools.usps.com/zip-code-lookup.htm?byaddress`, содержит страницу с формой для поиска почтового индекса по введенному адресу улицы (см. рис. 4.7). Чтобы воспользоваться этой формой в программе на Java, следует знать URL и параметры запроса по команде `POST`.

Эту информацию можно было бы получить, просмотрев код HTML-разметки формы, но ее, как правило, проще "выудить" из запроса с помощью сетевого монитора, входящего в набор инструментальных средств веб-разработки, предоставляемый в большинстве браузеров. В качестве примера на рис. 4.9 приведен моментальный снимок, сделанный сетевым монитором браузера Firefox при передаче на обработку данных выбранному для данного примера веб-сайту. На этом моментальном снимке можно выявить URL, параметры и значения, указанные при передаче данных на обработку.

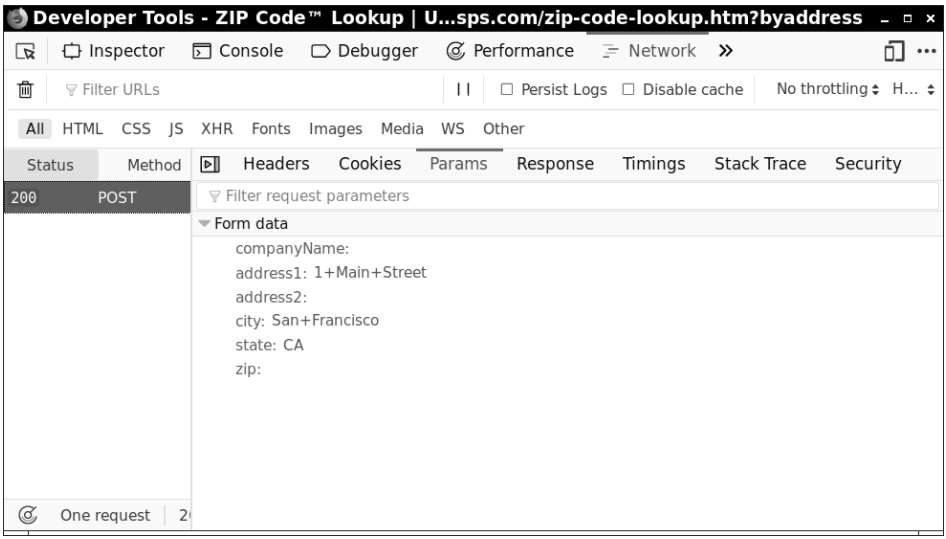


Рис. 4.9. Текущий контроль передачи HTML-формы на обработку

При передаче данных формы на обработку в HTTP-заголовок включается тип содержимого и его длина:

```
Content-Type: application/x-www-form-urlencoded
```

Данные можно передать и в других форматах. Так, если данные посылаются в формате JSON (JavaScript Object Notation — представление объектов JavaScript), в HTTP-заголовке запроса по команде `POST` должен быть указан тип содержимого `application/json`, а также его длина, как показано в следующем примере:

```
Content-Length: 124
```

В листинге 4.7 приведен исходный код примера программы, посылающей данные на сервер по команде `POST`. В файл свойств с расширением `.properties` вводятся следующие данные:

```
url=https://tools.usps.com/tools/app/ziplookup/zipByAddress
User-Agent=HTTPie/0.9.2
```

```
address1=1 Market Street
address2=
city=San Francisco
state=CA
companyName=
. . .
```

Программа удаляет элементы `url` и `User-Agent`, а все остальные элементы направляет методу `doPost()`. В методе `doPost()` сначала устанавливается соединение, а затем пользовательский посредник. (Служба определения почтовых индексов не обрабатывает устанавливаемый по умолчанию параметр запроса `User-Agent`, содержащий строку `"Java"`, вероятно, потому, что почтовая служба не намерена обслуживать программные запросы.)

Далее вызывается метод `setDoOutput(true)`, открывается поток вывода и перечисляются все ключи и значения. Для каждой пары "ключ–значение" по очереди передаются ключ, знак =, значение и разделительный знак `&`:

```
out.print(key);
out.print('=');
out.print(URLEncoder.encode(value, StandardCharsets.UTF_8));
if (дополнительные пары "ключ-значение") out.print('&');
```

Взаимодействие с сервером фактически происходит при переходе от записи к чтению любой части ответа. В заголовке `Content-Length` задается длина выводимых данных, а в заголовке `Content-Type` — тип `application/x-www-form-urlencoded`, если только не был указан другой тип содержимого. Заголовки и данные запроса посылаются серверу. Затем читаются заголовки и данные ответа сервера, которые могут быть запрошены. В данном примере программы такой переход от записи к чтению происходит при вызове метода `connection.getContentEncoding()`.

Следует, однако, иметь в виду, что если при выполнении серверной программы возникнет ошибка, то вызов метода `connection.getInputStream()` приведет к исключению типа `FileNotFoundException`. Тем не менее сервер продолжит передачу данных, отправив HTML-страницу с сообщением об ошибке. Обычно это сообщение `"Error 404-page not found"`, уведомляющее о том, что данная страница не найдена. Для фиксации страницы с этим сообщением об ошибке следует вызвать метод `getErrorStream()`:

```
InputStream err = connection.getErrorStream();
```

НА ЗАМЕТКУ! Метод **`getErrorStream()`**, а также ряд других методов, применяемых в рассматриваемом здесь примере программы, относятся к классу **`HttpURLConnection`**, производному от класса **`URLConnection`**. Если сделать запрос по URL, начинающемуся с префикса **`http://`** или **`https://`**, то полученный в итоге объект соединения можно привести к типу **`HttpURLConnection`**.

При передаче данных по команде `POST` на сервер серверная программа может *переадресовать* его по другому URL для получения искомой информации. Сервер может сделать это потому, что искомая информация находится в каком-нибудь другом месте. С другой стороны, он может предоставить отмеченный закладкой URL. Как правило, класс `HttpURLConnection` может осуществить переадресацию.

НА ЗАМЕТКУ! Если при переадресации требуется переслать cookie-файлы из одного сайта на другой, с этой целью можно настроить глобальный обработчик cookie-файлов следующим образом:

```
CookieHandler.setDefault(new CookieManager(null,
CookiePolicy.ACCEPT_ALL));
```

В этом случае cookie-файлы будут надлежащим образом включены в переадресацию.

Несмотря на то что переадресация, как правило, осуществляется автоматически, иногда это приходится делать вручную. Автоматическая переадресация между сетевыми протоколами HTTP и HTTPS не поддерживается из соображений безопасности. Она может не состояться и по менее ясным причинам. Например, в прежней версии службы определения почтовых индексов обычно применялась переадресация. Напомним, что параметр запроса `User-Agent` был установлен ранее таким образом, чтобы почтовая служба не посчитала, что запрос был сделан через прикладной интерфейс Java API. И хотя в первоначальном запросе можно задать другую строку для пользовательского посредника, такая настройка не используется при автоматической переадресации, при которой всегда посылается типичная строка пользовательского посредника, содержащая строку `"Java"`.

В подобных случаях переадресацию можно осуществить вручную. Прежде чем подключиться к серверу, необходимо выключить режим автоматической переадресации следующим образом:

```
connection.setInstanceFollowRedirects(false);
```

Сделав запрос, следует получить код ответа:

```
int responseCode = connection.getResponseCode();
```

и проверить, относится ли он к одному из перечисленных ниже кодов.

```
HttpURLConnection.HTTP_MOVED_PERM
HttpURLConnection.HTTP_MOVED_TEMP
HttpURLConnection.HTTP_SEE_OTHER
```

В таком случае следует сначала получить заголовок ответа `Location`, а затем URL для переадресации. Далее необходимо разорвать текущее соединение и установить другое соединение по новому URL, как показано ниже.

```
String location = connection.getHeaderField("Location");
if (location != null)
{
  URL base = connection.getURL();
  connection.disconnect();
  connection = (HttpURLConnection)
               new URL(base, location).openConnection();
  . . .
}
```

Приемы, демонстрируемые в рассматриваемой здесь программе, могут оказаться полезными всякий раз, когда требуется запросить информацию из существующего веб-сайта. Для этого достаточно выяснить сначала параметры, которые требуется послать в запросе, а затем удалить дескрипторы HTML-разметки и прочую ненужную информацию из полученного ответа.

Листинг 4.7. Исходный код из файла **post/PostTest.java**

```
1   package post;
2
3   import java.io.*;
4   import java.net.*;
5   import java.nio.charset.*;
6   import java.nio.file.*;
7   import java.util.*;
8
9   /**
10   * В этой программе демонстрируется применение
11   * класса URLConnection для формирования запроса
12   * по команде POST
13   * @version 1.42 2018-03-17
14   * @author Cay Horstmann
15   */
16  public class PostTest
17  {
18    public static void main(String[] args)
19          throws IOException
20    {
21      String propsFilename = args.length > 0 ? args[0]
22                           : "post/post.properties";
23      var props = new Properties();
24      try (InputStream in = Files.newInputStream(
25                           Paths.get(propsFilename)))
26      {
27        props.load(in);
28      }
29      String urlString = props.remove("url").toString();
30      Object userAgent = props.remove("User-Agent");
31      Object redirects = props.remove("redirects");
32      CookieHandler.setDefault(new CookieManager(null,
33                               CookiePolicy.ACCEPT_ALL));
34      String result = doPost(new URL(urlString), props,
35      userAgent == null ? null : userAgent.toString(),
36      redirects == null ? -1 : Integer.parseInt(
37                               redirects.toString()));
38      System.out.println(result);
39    }
40
41    /**
42     * Сделать HTTP-запрос по команде POST
43     * @param url Конкретный URL для отправки запроса
44     * @param nameValuePairs Параметры запроса
45     * @param userAgent Пользовательский посредник или
46     *                  пустое значение null, если это
47     *                  посредник по умолчанию
48     * @param redirects Количество последующих
49     *                  переадресаций вручную или
50     *                  значение -1, если переадресация
51     *                  производится автоматически
52     * @return Данные, возвращаемые из сервера
53     */
54    public static String doPost(URL url,
```

```
55                Map<Object, Object> nameValuePairs,
56                String userAgent, int redirects)
57          throws IOException
58    {
59      var connection = (HttpURLConnection)
60                       url.openConnection();
61      if (userAgent != null)
62        connection.setRequestProperty(
63                  "User-Agent", userAgent);
64      if (redirects >= 0)
65        connection.setInstanceFollowRedirects(false);
66
67      connection.setDoOutput(true);
68
69      try (var out = new PrintWriter(
70                     connection.getOutputStream()))
71      {
72        var first = true;
73        for (Map.Entry<Object, Object> pair :
74             nameValuePairs.entrySet())
75        {
76          if (first) first = false;
77          else out.print('&');
78          String name = pair.getKey().toString();
79          String value = pair.getValue().toString();
80          out.print(name);
81          out.print('=');
82          out.print(URLEncoder.encode(value,
83                    StandardCharsets.UTF_8));
84        }
85      }
86      String encoding = connection.getContentEncoding();
87      if (encoding == null) encoding = "UTF-8";
88
89      if (redirects > 0)
90      {
91        int responseCode = connection.getResponseCode();
92        if (responseCode == HttpURLConnection
93                            .HTTP_MOVED_PERM
94            || responseCode == HttpURLConnection
95                               .HTTP_MOVED_TEMP
96            || responseCode == HttpURLConnection
97                               .HTTP_SEE_OTHER)
98        {
99          String location = connection
100                           .getHeaderField("Location");
101         if (location != null)
102         {
103           URL base = connection.getURL();
104           connection.disconnect();
105           return doPost(new URL(base, location),
106                         nameValuePairs, userAgent,
107                         redirects - 1);
108         }
109       }
110     }
111     else if (redirects == 0)
```

```
112     {
113       throw new IOException("Too many redirects");
114     }
115
116     var response = new StringBuilder();
117     try (var in = new Scanner(
118          connection.getInputStream(), encoding))
119     {
120       while (in.hasNextLine())
121       {
122         response.append(in.nextLine());
123         response.append("\n");
124       }
125     }
126     catch (IOException e)
127     {
128       InputStream err = connection.getErrorStream();
129       if (err == null) throw e;
130       try (var in = new Scanner(err))
131       {
132         response.append(in.nextLine());
133         response.append("\n");
134       }
135     }
136
137     return response.toString();
138   }
139 }
```

java.net.HttpURLConnection 1.0

- **InputStream getErrorStream()**
  Возвращает поток ввода, из которого читаются сообщения сервера об ошибках.

java.net.URLEncoder 1.0

- **static String encode(String *s*, String *encoding*) 1.4**
  Возвращает строку ***s***, закодированную в формате URL с помощью заданной кодировки символов. (Рекомендуется указывать кодировку **"UTF-8"**.) При кодировании в формате URL символы **'A'-'Z'**, **'a'-'z'**, **'0'-'9'**, **'-'**, **'_'**, **'.'** и **'~'** оставляются без изменения. Пробелы заменяются знаками **'+'**, а все остальные символы — последовательностями кодированных байтов в форме **"%*XY*"**, где **0x*XY*** — шестнадцатеричное значение байта.

java.net.URLDecoder 1.2

- **static string decode(String *s*, String *encoding*) 1.4**
  Возвращает форму строки ***s***, закодированной в формате URL и декодированной с помощью заданной кодировки символов.

4.4. HTTP-клиент

Класс `URLConnection` был разработан еще до того, как сетевой протокол HTTP стал универсальным для Интернета. В этом классе поддерживается целый ряд сетевых протоколов, хотя поддержка протокола HTTP в нем реализована не очень удобно. Когда было принято решение о поддержке сетевого протокола HTTP/2, то стало ясно, что лучше предоставить современный клиентский интерфейс вместо того, чтобы переделывать уже существующий прикладной интерфейс API. Так, в классе `HttpClient` предоставляется более удобный прикладной интерфейс API для поддержки сетевого протокола HTTP/2. Начиная с версии Java 11 класс `HttpClient` входит в состав пакета `java.net.http`.

НА ЗАМЕТКУ! В версиях Java 9 и 10 прикладные программы следует запускать на выполнение из командной строки со следующим параметром:

```
--add-modules jdk.incubator.httpclient
```

В прикладном интерфейсе API для HTTP-клиента предоставляется более простой механизм подключения к веб-серверу, чем в классе `URLConnection`, где этот процесс дотошно выполняется в течение целого ряда стадий. HTTP-клиент, реализуемый средствами класса `HttpClient`, может выдавать запросы и получать ответы от веб-сервера. Чтобы получить такой клиент, достаточно сделать следующий вызов:

```
HttpClient client = HttpClient.newHttpClient()
```

Если требуется сконфигурировать клиент, то можно воспользоваться прикладным интерфейсом API его построителя, как показано ниже.

```
HttpClient client = HttpClient.newBuilder()
          .followRedirects(HttpClient.Redirect.ALWAYS)
          .build();
```

Подобным образом получается построитель, вызываются методы для специальной настройки создаваемого клиента, а затем вызывается метод `build()` с целью завершить весь процесс построения. Это типичный шаблон для построения неизменяемых объектов.

По такому же шаблону построения составляются запросы. В качестве примера ниже демонстрируется составление HTTP-запроса по команде `GET`.

```
HttpRequest request = HttpRequest.newBuilder()
            .uri(new URI("http://horstmann.com"))
            .GET()
            .build();
```

Универсальный идентификатор ресурса (URI) в сетевом протоколе HTTP равнозначен URL. Но в Java предоставляется класс `URL` с методами, фактически устанавливающими соединение с веб-ресурсом по заданному URL, тогда как класс `URI` обеспечивает лишь необходимый синтаксис (схему, хост, порт, путь, запрос, фрагмент и т.д.).

Для составления HTTP-запроса по команде `POST` требуется "издатель тела запроса", где запрашиваемые данные преобразуются в пересылаемые данные. Имеются издатели тела запроса для символьных строк, массивов байтов и файлов.

Так, если запрос составляется в формате JSON, издателю его тела достаточно предоставить символьную строку в формате JSON, как показано ниже.

```
HttpRequest request = HttpRequest.newBuilder()
            .uri(new URI(url))
            .header("Content-Type", "application/json")
            .POST(HttpRequest.BodyPublishers
            .ofString(jsonString))
            .build();
```

К сожалению, в рассматриваемом здесь прикладном интерфейсе API не поддерживается требующееся форматирование общеупотребительных типов содержимого запросов. В приведенном далее примере программы из листинга 4.8 демонстрируется применение издателей тела запроса для обработки данных формы и выгрузки файлов.

Отправляя запрос на веб-сервер, приходится указывать клиенту порядок обработки получаемого ответа. Если же требуется отравить лишь тело запроса в виде символьной строки, это можно сделать с помощью метода `HttpResponse.BodyHandlers.ofString()` следующим образом:

```
HttpResponse<String> response = client.send(request,
                     HttpResponse.BodyHandlers.ofString());
```

Класс `HttpResponse` является обобщенным, а параметр его типа обозначает тип тела запроса. Получить тело запроса в виде символьной строки можно следующим образом:

```
String bodyString = response.body();
```

Имеются и другие обработчики тела ответа, получающие ответ в виде массива байтов или потока ввода. В частности, метод `BodyHandlers.ofFile(filePath)` возвращает обработчик, сохраняющий ответ в заданном файле, а метод `BodyHandlers.ofFileDownload(directoryPath)` сохраняет ответ в заданном каталоге, используя имя файла из заголовка `Content-Disposition`. Наконец, обработчик, возвращаемый из метода `BodyHandlers.discarding()`, просто отвергает полученный ответ.

Обработка содержимого ответа не обеспечивается как составная часть рассматриваемого здесь прикладного интерфейса API. Так, если ответ получается в формате JSON, для синтаксического анализа его содержимого потребуется отдельная библиотека, поддерживающая обработку данных формата JSON.

В объекте типа `HttpResponse` предоставляется также код состояния и заголовки ответов:

```
int status = response.statusCode();
HttpHeaders responseHeaders = response.headers();
```

Объекты типа `HttpHeaders` можно преобразовать в отображение, как демонстрируется в приведенной ниже строке кода. В качестве значений в таком отображении служат списки, поскольку в сетевом протоколе HTTP для каждого ключа допускается несколько значений.

```
Map<String, List<String>> headerMap = responseHeaders.map();
```

Если же требуется значение для конкретного ключа и заранее известно, что у него не может быть несколько значений, следует вызвать метод `firstValue()`,

как показано ниже. В ответ получается конкретное значение, а если оно не предоставлено — пустое необязательное значение.

```
Optional<String> lastModified =
                headerMap.firstValue("Last-Modified");
```

Ответы можно обрабатывать и асинхронно. Для этого при построении клиента предоставляется исполнитель:

```
ExecutorService executor = Executors.newCachedThreadPool();
HttpClient client = HttpClient.newBuilder()
                  .executor(executor).build();
```

Сначала составляется запрос, а затем для клиента вызывается метод `sendAsync()`, как показано ниже. В итоге получается завершаемое будущее действие типа `CompletableFuture<HttpResponse<T>>`, где `T` — тип обработчика тела ответа. О том, как применять прикладной интерфейс API для завершаемых будущих действий, см. в главе 12 первого тома настоящего издания.

```
HttpRequest request = HttpRequest.newBuilder().uri(uri)
                                  .GET().build();
client.sendAsync(request,
                HttpResponse.BodyHandlers.ofString())
    thenAccept(response -> . . .);
```

СОВЕТ. Чтобы активизировать режим протоколирования для HTTP-клиента типа **`HttpClient`**, достаточно ввести следующую строку кода в файл **`net.properties`** свойств комплекта JDK:

`jdk.httpclient.HttpClient.log=all`

Вместо параметра **`all`** можно указать разделяемый запятыми список параметров **`headers`**, **`requests`**, **`content`**, **`errors`**, **`ssl`**, **`trace`** и **`frames`**, а после них дополнительно — параметры **`:control`**, **`:data`**, **`:window`** или **`:all`**, но без пробелов.

После этого можно установить уровень протоколирования **`INFO`** для регистратора **`jdk.httpclient.HttpClient`**, введя, например, следующую строку кода в файл **`logging.properties`** свойств комплекта JDK:

`jdk.httpclient.HttpClient.level=INFO`

Листинг 4.8. Исходный код из файла **`client/HttpClientTest.java`**

```
1   package client;
2
3   import java.io.*;
4   import java.math.*;
5   import java.net.*;
6   import java.nio.charset.*;
7   import java.nio.file.*;
8   import java.util.*;
9
10  import java.net.http.*;
11   import java.net.http.HttpRequest.*;
12
13  class MoreBodyPublishers
14  {
15    public static BodyPublisher ofFormData(
```

```
16                  Map<Object, Object> data)
17    {
18      var first = true;
19      var builder = new StringBuilder();
20      for (Map.Entry<Object, Object> entry :
21           data.entrySet())
22      {
23        if (first) first = false;
24        else builder.append("&");
25        builder.append(URLEncoder.encode(
26                       entry.getKey().toString(),
27                       StandardCharsets.UTF_8));
28        builder.append("=");
29        builder.append(URLEncoder.encode(
30                       entry.getValue().toString(),
31                       StandardCharsets.UTF_8));
32      }
33      return BodyPublishers.ofString(builder.toString());
34    }
35
36    private static byte[] bytes(String s)
37      { return s.getBytes(StandardCharsets.UTF_8); }
38
39    public static BodyPublisher ofMimeMultipartData(
40           Map<Object, Object> data, String boundary)
41           throws IOException
42    {
43      var byteArrays = new ArrayList<byte[]>();
44      byte[] separator = bytes("--" + boundary
45            + "\nContent-Disposition: form-data; name=");
46      for (Map.Entry<Object, Object> entry :
47           data.entrySet())
48      {
49        byteArrays.add(separator);
50
51        if (entry.getValue() instanceof Path)
52        {
53          var path = (Path) entry.getValue();
54          String mimeType = Files.probeContentType(path);
55          byteArrays.add(bytes("\"" + entry.getKey()
56                 + "\"; filename=\"" + path.getFileName()
57                 + "\"\nContent-Type: "
58                 + mimeType + "\n\n"));
59          byteArrays.add(Files.readAllBytes(path));
60        }
61        else
62          byteArrays.add(bytes("\"" + entry.getKey()
63              + "\"\n\n" + entry.getValue() + "\n"));
64      }
65      byteArrays.add(bytes("--" + boundary + "--"));
66      return BodyPublishers.ofByteArrays(byteArrays);
67    }
68
69    public static BodyPublisher ofSimpleJSON(
70                  Map<Object, Object> data)
71    {
72      var builder = new StringBuilder();
```

```
73      builder.append("{");
74      var first = true;
75      for (Map.Entry<Object, Object> entry :
76           data.entrySet())
77      {
78        if (first) first = false;
79        else
80          builder.append(",");
81        builder.append(jsonEscape(entry.getKey()
82                       .toString()))
83              .append(": ")
83              .append(jsonEscape(entry.getValue()
84                       .toString()));
85      }
86      builder.append("}");
87      return BodyPublishers.ofString(builder.toString());
88    }
89
90    private static Map<Character, String> replacements =
91            Map.of('\b', "\\b", '\f', "\\f", \n', '"\\n",
92                   '\r', "\\r", '\t', "\\t", '"', "\\"",
93                   '\\', "\\\\");
94
95    private static StringBuilder jsonEscape(String str)
96    {
97      var result = new StringBuilder("\"");
98      for (int i = 0; i < str.length(); i++)
99      {
100       char ch = str.charAt(i);
101       String replacement = replacements.get(ch);
102       if (replacement == null) result.append(ch);
103       else result.append(replacement);
104     }
105     result.append("\"");
106     return result;
107   }
108 }
109
110 public class HttpClientTest
111 {
112   public static void main(String[] args)
113           throws IOException, URISyntaxException,
114                  InterruptedException
115   {
116     System.setProperty("jdk.httpclient.HttpClient.log",
117                        "headers,errors");
118     String propsFilename = args.length > 0 ? args[0] :
119                            "client/post.properties";
120     Path propsPath = Paths.get(propsFilename);
121     var props = new Properties();
122     try (InputStream in =
123          Files.newInputStream(propsPath))
124     {
126       props.load(in);
127     }
128     String urlString = "" + props.remove("url");
129     String contentType = ""
```

```
130          + props.remove("Content-Type");
131     if (contentType.equals("multipart/form-data"))
132     {
133       var generator = new Random();
134       String boundary = new BigInteger(256, generator)
135                             .toString();
136       contentType += ";boundary=" + boundary;
137       props.replaceAll((k, v) ->
138             v.toString().startsWith("file://")
139                ? propsPath.getParent()
140                  .resolve(Paths.get(v.toString()
141                  .substring(7)))
142                : v);
143     }
144     String result = doPost(urlString,
145                            contentType, props);
146      System.out.println(result);
147   }
148
149   public static String doPost(String url,
150          String contentType, Map<Object, Object> data)
151          throws IOException, URISyntaxException,
152                 InterruptedException
153   {
154     HttpClient client = HttpClient.newBuilder()
155        .followRedirects(HttpClient.Redirect.ALWAYS)
156        .build();
157
158     BodyPublisher publisher = null;
159     if (contentType.startsWith("multipart/form-data"))
160     {
161       String boundary = contentType.substring(
162              contentType.lastIndexOf("=") + 1);
163       publisher = MoreBodyPublishers
164                   .ofMimeMultipartData(data, boundary);
165     }
166     else if (contentType.equals(
167              "application/x-www-form-urlencoded"))
168     publisher = MoreBodyPublishers.ofFormData(data);
169     else
170     {
171       contentType = "application/json";
172       publisher = MoreBodyPublishers.ofSimpleJSON(data);
173     }
174
175     HttpRequest request = HttpRequest.newBuilder()
176                 .uri(new URI(url))
177                 .header("Content-Type", contentType)
178                 .POST(publisher)
179                 .build();
180     HttpResponse<String> response = client.send(
181         request, HttpResponse.BodyHandlers.ofString());
182     return response.body();
183   }
184 }
```

java.net.http.HttpClient 11

- **static HttpClient newHttpClient()**
  Возвращает объект типа **HttpClient** с конфигурацией HTTP-клиента по умолчанию.
- **static HttpClient.Builder newBuilder()**
  Возвращает построитель HTTP-клиентов, представленных объектами типа **HttpClient**.
- **<T> HttpResponse<T> send(HttpRequest *request*, HttpResponse.BodyHandler<T> *responseBodyHandler*)**
- **<T> CompletableFuture<HttpResponse<T>> sendAsync(HttpRequest *request*, HttpResponse .BodyHandler<T> *responseBodyHandler*)**
  Составляют синхронный и асинхронный запрос и обрабатывают тело получаемого ответа с помощью заданного обработчика.

java.net.http.HttpClient.Builder 11

- **HttpClient build()**
  Возвращает объект типа **HttpClient** со свойствами, сконфигурированными данным построителем HTTP-клиентов.
- **HttpClient.Builder followRedirects(HttpClient.Redirect *policy*)**
  Устанавливает правило переадресации, определяемое одной из следующих констант из перечисления **HttpClient.Redirect**: **ALWAYS**, **NEVER** или **NORMAL** (отклонять переадресацию только из сетевого протокола HTTPS в протокол HTTP).
- **HttpClient.Builder executor(Executor *executor*)**
  Устанавливает исполнитель асинхронных запросов.

java.net.http.HttpRequest 11

- **HttpRequest.Builder newBuilder()**
  Возвращает построитель HTTP-запросов, представленных объектами типа **HttpRequest**.

java.net.http.HttpRequest.Builder 11

- **HttpRequest build()**
  Возвращает объект типа **HttpRequest** со свойствами, сконфигурированными данным построителем HTTP-запросов.
- **HttpRequest.Builder uri(URI *uri*)**
  Устанавливает URI для данного запроса.
- **HttpRequest.Builder header(String *name*, String *value*)**
  Устанавливает заголовок для данного запроса.

java.net.http.HttpRequest.Builder 11 *(окончание)*

- **HttpRequest.Builder GET()**
- **HttpRequest.Builder DELETE()**
- **HttpRequest.Builder POST(HttpRequest.BodyPublisher *bodyPublisher*)**
- **HttpRequest.Builder PUT(HttpRequest.BodyPublisher *bodyPublisher*)**

  Устанавливают метод доступа и тело для данного запроса.

java.net.http.HttpResponse<T> 11

- **T body()**

  Возвращает тело данного ответа.
- **int statusCode()**

  Возвращает код состояния для данного ответа.
- **HttpHeaders headers()**

  Возвращает заголовки ответа.

java.net.http.HttpHeaders 11

- **Map<String,List<String>> map()**

  Возвращает отображение типа **Map** данных заголовков.
- **Optional<String> firstValue(String *name*)**

  Возвращает первое значение по имени, указанному в данных заголовках, если таковое имеется.

4.5. Отправка электронной почты

В прошлом для отправки электронной почты достаточно было написать программу, установливавшую соединение с сетевым сокетом через порт 25, который обычно используется для работы сетевого протокола SMTP (Simple Mail Transport Protocol — простой протокол передачи почты), описывающего формат электронных сообщений. После подключения к серверу в данной программе нужно было послать заголовок сообщения, который достаточно просто было создать в формате SMTP, а затем и текст сообщения, выполнив перечисленные ниже действия.

1. Открыть сокет на своем компьютере, подключенном к Интернету, как показано ниже.

```
var s = new Socket("mail.yourserver.com", 25);
    // номер порта 25 соответствует протоколу SMTP
var out = new PrintWriter(s.getOutputStream(),"UTF-8");
```

2. Направить в поток вывода следующие данные:

```
HELO хост отправителя
MAIL FROM: адрес отправителя
```

```
RCPT TO: адрес получателя
DATA
Subject: тема
(пустая строка)
почтовое сообщение
(любое количество строк)

QUIT
```

В спецификации сетевого протокола SMTP (документ RFC 821) требуется, чтобы строки завершались последовательностями символов /r и /n. Первоначально SMTP-серверы исправно направляли электронную почту от любого адресата. Но когда навязчивые сообщения наводнили Интернет, большинство этих серверов было оснащено встроенными проверками и принимали запросы только по тем IP-адресам, которым они доверяют. Аутентификация обычно происходит через безопасные сокетные соединения.

Реализовать алгоритмы подобной аутентификации вручную — дело непростое. Поэтому в этом разделе будет показано, как пользоваться прикладным интерфейсом JavaMail API для отправки сообщений электронной почты из программы на Java. С этой целью загрузите данный прикладной интерфейс по адресу `https://javaee.github.io/javamail/` и разархивируйте его на жесткий диск своего компьютера.

Чтобы воспользоваться прикладным интерфейсом JavaMail API, необходимо установить некоторые свойства, зависящие от конкретного почтового сервера. В качестве примера ниже приведены свойства, устанавливаемые для почтового сервера GMail. Они считываются из файла свойств в рассматриваемом здесь примере программы из листинга 4.9.

```
mail.transport.protocol=smtps
mail.smtps.auth=true
mail.smtps.host=smtp.gmail.com
mail.smtps.user=cayhorstmann@gmail.com
```

Из соображений безопасности пароль не вводится в файл свойств и предлагается для ввода вручную. После чтения из файла свойств сеанс почтовой связи устанавливается следующим образом:

```
Session mailSession = Session.getDefaultInstance(props);
```

Затем составляется почтовое сообщение с указанием требуемого отправителя, получателя, темы и текста самого сообщения:

```
MimeMessage message = new MimeMessage(mailSession);
message.setFrom(new InternetAddress(from));
message.addRecipient(RecipientType.TO,
                     new InternetAddress(to));
message.setSubject(subject);
message.setText(builder.toString());
```

Далее почтовое сообщение отправляется следующим образом:

```
Transport tr = mailSession.getTransport();
tr.connect(null, password);
tr.sendMessage(message, message.getAllRecipients());
tr.close();
```

Рассматриваемая здесь программа читает почтовое сообщение из текстового файла в приведенном ниже формате.

```
Отправитель
Получатель
Тема
Текст сообщения (любое количество строк)
```

Кроме упомянутого выше прикладного интерфейса JavaMail API, для выполнения данной программы потребуется архивный JAR-файл каркаса JavaBeans Activation Framework, который можно загрузить по адресу `https://www.oracle.com/technetwork/java/javase/downloads/index-135046.html#download` или из центрального хранилища Maven Central по адресу `https://mvnrepository.com/artifact/javax.activation/activation`. Затем выполните следующую команду:

```
java -classpath .:javax.mail.jar:activation-1.1.1.jar
     path/to/message.txt
```

На момент написания данной книги почтовый сервер GMail не проверял достоверность получаемой информации, а следовательно, в почтовом сообщении можно было указать любого отправителя. (Это обстоятельство следует иметь в виду при получении от отправителя по адресу `president@whitehouse.gov` очередного приглашения на официальный прием, организуемый на лужайке перед Белым домом.)

СОВЕТ. Если вам не удастся выяснить причину, по которой соединение с почтовым сервером не действует, сделайте следующий вызов и проверьте почтовые сообщения:

```
mailSession.setDebug(true);
```

Кроме того, обратитесь за полезными советами на веб-страницу JavaMail API FAQ (Часто задаваемые вопросы по прикладному программному интерфейсу JavaMail API FAQ), доступную по адресу `https://javaee.github.io/javamail/FAQ`.

Листинг 4.9. Исходный код из файла **`mail/MailTest.java`**

```
1  package mail;
2
3  import java.io.*;
4  import java.nio.charset.*;
5  import java.nio.file.*;
6  import java.util.*;
7  import javax.mail.*;
8  import javax.mail.internet.*;
9  import javax.mail.internet.MimeMessage.RecipientType;
10
11 /**
12  * В этой программе демонстрируется применение
13  * прикладного интерфейса JavaMail API для отправки
14  * сообщений по электронной почте
15  * @author Cay Horstmann
16  * @version 1.01 2018-03-17
17  */
18 public class MailTest
19 {
```

```
20   public static void main(String[] args)
21         throws MessagingException, IOException
22   {
23     var props = new Properties();
24     try (InputStream in = Files.newInputStream(
25          Paths.get("mail", "mail.properties")))
26     {
27        props.load(in);
28     }
29     List<String> lines = Files.readAllLines(
30          Paths.get(args[0]), StandardCharsets.UTF_8);
31
32     String from = lines.get(0);
33     String to = lines.get(1);
34     String subject = lines.get(2);
35
36     var builder = new StringBuilder();
37     for (int i = 3; i < lines.size(); i++)
38     {
39       builder.append(lines.get(i));
40       builder.append("\n");
41     }
42
43     Console console = System.console();
44     var password =
45         new String(console.readPassword("Password: "));
46
47     Session mailSession =
48            Session.getDefaultInstance(props);
49     // mailSession.setDebug(true);
50     var message = new MimeMessage(mailSession);
51     message.setFrom(new InternetAddress(from));
52     message.addRecipient(RecipientType.TO,
53                          new InternetAddress(to));
54     message.setSubject(subject);
55     message.setText(builder.toString());
56     Transport tr = mailSession.getTransport();
57     try
58     {
59       tr.connect(null, password);
60       tr.sendMessage(message,
61                      message.getAllRecipients());
62     }
63     finally
64     {
65       tr.close();
66     }
67   }
68 }
```

В этой главе было показано, как на Java пишется исходный код программ для сетевых клиентов и серверов и как организуется сбор данных с веб-серверов. В следующей главе речь пойдет о взаимодействии с базами данных. Из нее вы узнаете, как работать с реляционными базами данных в программах на Java, используя прикладной интерфейс JDBC API.

ГЛАВА 5

Работа с базами данных

В этой главе...

- Структура JDBC
- Язык SQL
- Конфигурирование JDBC
- Работа с операторами JDBC
- Выполнение запросов
- Прокручиваемые и обновляемые результирующие наборы
- Наборы строк
- Метаданные
- Транзакции
- Расширенные типы данных SQL
- Управление подключением к базам данных в веб- и корпоративных приложениях

В 1996 году компания Sun Microsystems выпустила первую версию прикладного интерфейса API для организации доступа из программ на Java к базам данных (JDBC). Этот прикладной интерфейс позволяет соединяться с базой данных, запрашивать и обновлять данные с помощью языка структурированных запросов (Structured Query Language — SQL). Язык SQL фактически стал стандартным средством взаимодействия с реляционными базами данных. С тех пор JDBC стал одним из наиболее употребительных прикладных интерфейсов API в библиотеке Java.

Прикладной интерфейс JDBC неоднократно обновлялся. На момент написания данной книги самой последней считалась версия JDBC 4.3, включенная в состав версии Java 9.

В этой главе рассматриваются принципы, положенные в основу прикладного интерфейса JDBC. Из нее вы узнаете (а возможно, лишь вспомните) о языке SQL,

который является стандартным средством доступа к реляционным базам данных. В ней будут также рассмотрены примеры применения интерфейса JDBC, демонстрирующие наиболее распространенные приемы обращения с базами данных в прикладных программах.

НА ЗАМЕТКУ! Как заявляют в компании Oracle, JDBC — это торговая марка, а не сокращение Java Database Connectivity. Она была придумана по аналогии с обозначением ODBC стандартного прикладного интерфейса для работы с базами данных, который был первоначально предложен корпорацией Microsoft и затем внедрен в стандарт SQL.

5.1. Структура JDBC

Создатели Java с самого начала осознавали потенциальные преимущества данного языка для работы с базами данных. С 1995 года они начали работать над расширением стандартной библиотеки Java для организации доступа к базам данных средствами SQL. Сначала они попробовали создать такие расширения Java, которые позволили бы осуществлять доступ к произвольной базе данных *только* средствами Java, но очень скоро убедились в бесперспективности такого подхода, поскольку для доступа к базам данных применялись самые разные протоколы. Кроме того, поставщики программного обеспечения баз данных были весьма заинтересованы в разработке на Java стандартного сетевого протокола для доступа к базам данных, но при условии, что за основу будет принят их *собственный* сетевой протокол.

В конечном счете поставщики баз данных и инструментальных средств для доступа к ним сошлись на том, что лучше предоставить прикладной интерфейс API только на Java для доступа к базам данных средствами SQL, а также диспетчер драйверов, который позволил бы подключать к базам драйверы независимых производителей. Такой подход позволял поставщикам баз данных создавать собственные драйверы, которые подключались бы с помощью данного диспетчера. Предполагалось, что это будет простой механизм регистрации сторонних драйверов.

Подобная организация прикладного интерфейса JDBC основана на весьма удачной модели интерфейса ODBC, разработанного в корпорации Microsoft. В основу интерфейсов JDBC и ODBC положен общий принцип: программы, написанные в соответствии с требованиями прикладного интерфейса API, способны взаимодействовать с диспетчером драйверов JDBC, который, в свою очередь, использует подключаемые драйверы для обращения к базе данных. Это означает, что для работы с базами данных в прикладных программах достаточно пользоваться средствами JDBC API.

5.1.1. Типы драйверов JDBC

Каждый драйвер JDBC относится к одному из перечисленных ниже типов.

- **Драйвер типа 1.** Преобразует интерфейс JDBC в ODBC и для взаимодействия с базой данных использует драйвер ODBC. Один такой драйвер был включен в первые версии Java под названием *мост JDBC/ODBC*. Но для его применения требуется установить и настроить соответствующим образом драйвер ODBC. В первом выпуске JDBC этот мост предполагалось использовать только для тестирования, а не для применения в рабочих программах.

В настоящее время уже имеется достаточное количество более удачных драйверов, поэтому пользоваться мостом JDBC/ODBC не рекомендуется.

- **Драйвер типа 2.** Написан частично на Java и отчасти использует платформенно-ориентированный код для взаимодействия с клиентским прикладным интерфейсом API базы данных. Для применения такого драйвера, помимо библиотеки Java, на стороне клиента необходимо установить код, специфический для конкретной платформы.
- **Драйвер типа 3.** Разрабатывается только на основе клиентской библиотеки Java, в которой используется независимый от базы данных протокол передачи запросов базы данных на сервер. Этот протокол приводит запросы базы данных в соответствие с характерным для нее протоколом. Развертывание прикладных программ значительно упрощается благодаря тому, что код, зависящий от конкретной платформы, находится только на сервере.
- **Драйвер типа 4.** Представляет собой библиотеку, написанную только на Java, для приведения запросов JDBC в соответствие с протоколом конкретной базы данных.

НА ЗАМЕТКУ! Спецификация прикладного интерфейса JDBC доступна для загрузки по адресу `https://jcp.org/en/jsr/detail?id=221`.

Большинство поставщиков баз данных предоставляют драйверы типа 3 или 4. Кроме того, целый ряд сторонних производителей специализируется на создании драйверов, которые позволяют добиться более полного соответствия принятым стандартам, поддерживают большее количество платформ, обладают более высокой производительностью или надежностью, чем драйверы, предлагаемые поставщиками баз данных.

Основные цели прикладного интерфейса JDBC можно сформулировать следующим образом.

- Разработчики пишут программы на Java, пользуясь для доступа к базам данных стандартными средствами языка SQL (или его специализированными расширениями), но следуя только соглашениям, принятым в Java.
- Поставщики баз данных и инструментальных средств к ним предоставляют драйверы только низкого уровня. Это дает им возможность оптимизировать драйверы под свою конкретную продукцию.

НА ЗАМЕТКУ! На конференции JavaOne в мае 1996 года представители компании Sun Microsystems указали на ряд следующих причин отказа от модели ODBC.

- Трудна в освоении.
- Имеет всего лишь несколько команд с большим количеством параметров, тогда как стиль программирования на Java основан на применении большого количества простых и интуитивно понятных методов.
- Основана на использовании указателей типа `void*` и других элементов языка C, отсутствующих в Java.
- Менее безопасна и более сложна для развертывания, чем решение, получаемое только на Java.

5.1.2. Типичные примеры применения JDBC

Согласно традиционной модели "клиент–сервер" графический пользовательский интерфейс (ГПИ) реализуется на стороне клиента, а база данных располагается на стороне сервера (рис. 5.1). В этом случае драйвер JDBC развертывается на стороне клиента.

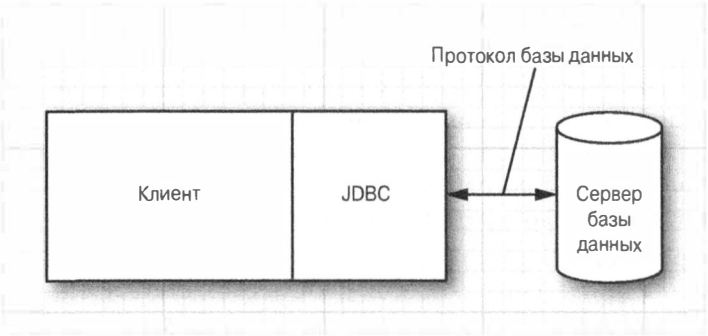


Рис. 5.1. Традиционная структура приложений "клиент-сервер"

Но в настоящее время существует явная тенденция к переходу от архитектуры "клиент–сервер" к трехуровневой модели или даже более совершенной *n*-уровневой модели. В трехуровневой модели клиент не формирует обращения к базе данных. Вместо этого он обращается к средствам промежуточного уровня на сервере, который, в свою очередь, выполняет запросы к базе данных. Трехуровневая модель обладает двумя преимуществами: отделяет *визуальное представление* (на клиентском компьютере) от *бизнес-логики* (на промежуточном уровне) и *исходных данных* (хранящихся в базе данных). Таким образом, становится возможным доступ к тем же самым данным по одинаковым бизнес-правилам со стороны разнотипных клиентов, в том числе прикладных программ на Java, веб-браузеров и приложений для мобильных устройств.

Взаимодействие между клиентом и промежуточным уровнем может быть реализовано по сетевому протоколу HTTP. А прикладной интерфейс JDBC служит для управления взаимодействием между промежуточным уровнем и серверной базой данных. На рис. 5.2 схематически показана основная архитектура трехуровневой модели.

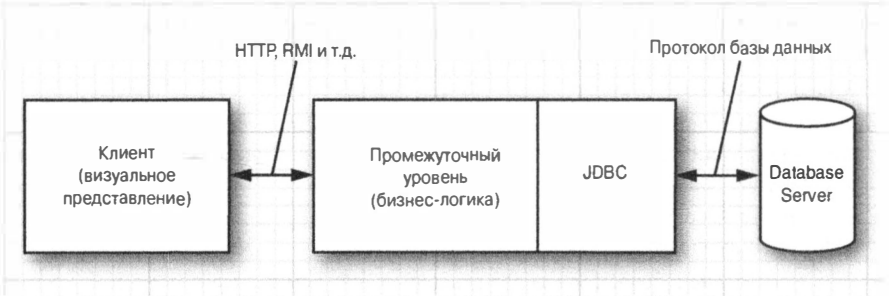


Рис. 5.2. Структура приложений на основе трехуровневой модели

5.2. Язык SQL

Прикладной интерфейс JDBC позволяет взаимодействовать с базами данных посредством языка SQL, который, в свою очередь, образует интерфейс для большинства современных реляционных баз данных. Настольные базы данных предоставляют графический интерфейс, который дает пользователям возможность непосредственно манипулировать данными, но доступ к серверным базам данных возможен только средствами языка SQL.

Пакет JDBC можно рассматривать лишь как прикладной интерфейс API для взаимодействия с операторами языка SQL с целью получить доступ к базам данных. В этом разделе приводится краткое описание языка SQL. Если вам не приходилось раньше иметь дело с SQL, то сведений, представленных в этом разделе, может оказаться недостаточно. Для более досконального изучения основ SQL можно порекомендовать книгу *Learning SQL* Алана Болью (Alan Beaulieu; издательство O'Reilly, 2009 г.) или *Learn SQL The Hard Way* Зеда А. Шоу (Zed A. Shaw), оперативно доступную в электронном виде для заказа по адресу `http://sql.learncodethehardway.org/`.

База данных представляет собой набор именованных таблиц со строками и столбцами. Каждый столбец имеет свое *имя*, а данные хранятся в строках. В качестве примера базы данных здесь и далее рассматривается ряд таблиц с описаниями библиотеки классических книг по вычислительной технике (табл. 5.1–5.4).

Таблица 5.1. Таблица `Authors`

Author_ID	Name	Fname
ALEX	Alexander	Christopher
BROO	Brooks	Frederick P.
...	...	...

Таблица 5.2. Таблица `Books`

Title	ISBN	Publisher_ID	Price
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
...	...	...	

Таблица 5.3. Таблица `BooksAuthors`

ISBN	Author_ID	Seq_No
0-201-96426-0	DATE	1
0-201-96426-0	DARW	2
0-19-501919-9	ALEX	1
...	...	...

Таблица 5.4. Таблица `Publishers`

Publisher_ID	Name	URL
0201	Addison-Wesley	www.aw-bc.com
0407	John Wiley & Sons	www.wiley.com
...	...	...

На рис. 5.3 представлена таблица `Books`, а на рис. 5.4 — результат *соединения* таблиц `Books` и `Publishers`. Обе таблицы содержат идентификатор издателя. При соединении таблиц по этому идентификатору получается *результат запроса* в виде таблицы, содержащей данные из обеих исходных таблиц. В каждой строке этой таблицы содержатся сведения о книге, название и адрес веб-сайта издательства. Обратите внимание на то, что данные с названием книги и адресом веб-сайта неоднократно дублируются, поскольку в результирующей таблице оказывается несколько строк, относящихся к одному и тому же издательству.

Преимущество соединения таблиц заключается в том, что при этом удается избежать нежелательного дублирования данных. Например, в простейшей структуре базы данных таблица `Books` может содержать столбцы с названием и адресом веб-сайта издательства. Но в таком случае данные будут дублироваться уже не только в результате запроса, но и в самой базе данных.

При изменении адреса веб-сайта придется также изменить эти данные во *всех* записях в базе данных. Очевидно, что при выполнении столь трудоемкой задачи могут легко возникнуть ошибки. В реляционной модели данные распределяются среди нескольких таблиц таким образом, чтобы они не дублировались без особой надобности. Например, адрес веб-сайта каждого издательства хранится в единственном экземпляре в таблице с данными об издательствах. При необходимости данные из разных таблиц нетрудно соединить в результат запроса.

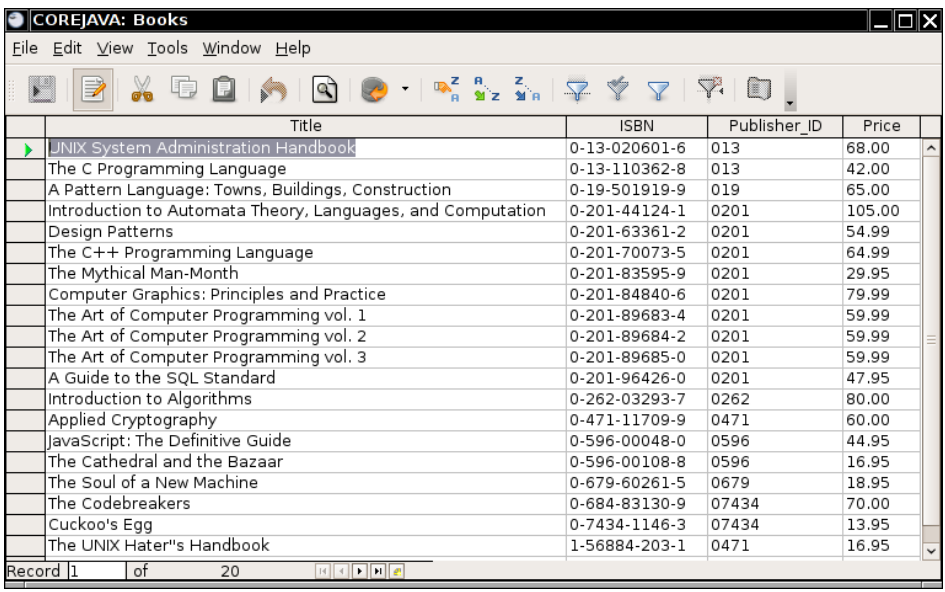


Рис. 5.3. Образец таблицы с данными о книгах

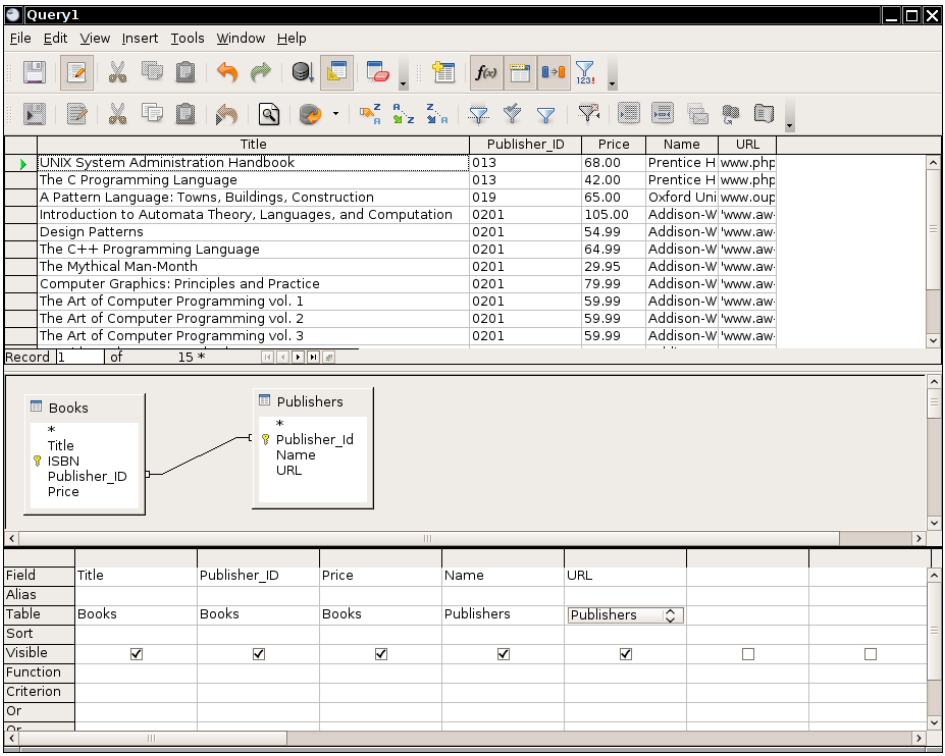


Рис. 5.4. Результат соединения двух таблиц

На рис. 5.3 и 5.4 показано графическое инструментальное средство, предназначенное для просмотра и связывания таблиц. Многие поставщики программного обеспечения предлагают разнообразные диалоговые инструментальные средства для создания запросов путем манипулирования столбцами и ввода данных в готовые формы. Они называются инструментальными средствами составления *запросов по образцу* (QBE). А при использовании SQL запрос создается в текстовом виде в строгом соответствии с синтаксисом этого языка, как показано ниже.

```
SELECT Books.Title, Books.Publisher_Id, Books.Price,
       Publishers.Name, Publishers.URL
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

В оставшейся части этого раздела описываются основные способы создания подобных запросов базы данных. Читатели, знакомые с SQL, могут пропустить этот материал. Ключевые слова SQL принято вводить прописными буквами, хотя это правило не является обязательным.

Оператор `SELECT` может применяться в самых разных целях, например, для выбора всех элементов из таблицы `Books` по следующему запросу:

```
SELECT * FROM Books
```

Предложение `FROM` обязательно указывается в каждом операторе `SELECT`. В этом предложении базе данных сообщается о тех таблицах, в которых

требуется выполнить поиск данных. В операторе SELECT можно указать любые требующиеся столбцы следующим образом:

```
SELECT ISBN, Price, Title
FROM Books
```

Выбор строк можно ограничить с помощью условия, указываемого в предложении WHERE:

```
SELECT ISBN, Price, Title
FROM Books
WHERE Price <= 29.95
```

Особо следует подчеркнуть, что для сравнения в SQL используются операции = и <>, а не == или !=, как при программировании на Java.

НА ЗАМЕТКУ! Некоторые поставщики баз данных используют операцию **!=** для обозначения сравнения, но учтите, что такое обозначение не соответствует стандарту SQL, поэтому пользоваться им не рекомендуется.

В предложении WHERE может присутствовать операция LIKE для сопоставления с заданным шаблоном. Но вместо обычных символов подстановки * и ? в данной операции употребляется знак %, обозначающий любое количество символов, а знак _ — один символ. Ниже приведен пример запроса на выборку книг, в названиях которых отсутствует такое слово, как UNIX или Linux.

```
SELECT ISBN, Price, Title
FROM Books
WHERE Title NOT LIKE '%n_x%'
```

Обратите внимание на то, что в запросах базы данных символьные строки заключаются в одиночные, а не в двойные кавычки. Одиночная кавычка в символьной строке обозначается парой одиночных кавычек, как в приведенном ниже примере запроса на поиск всех книг, в названиях которых содержится одиночная кавычка.

```
SELECT Title
FROM Books
WHERE Title LIKE '%''%'
```

Чтобы выбрать данные из нескольких таблиц, их нужно перечислить в следующем порядке:

```
SELECT * FROM Books, Publishers
```

Но без предложения WHERE такой запрос не представляет большого интереса, поскольку по нему получаются все сочетания строк из обеих таблиц. В данном случае таблица Books содержит 20 строк, а таблица Publishers — 8 строк. Поэтому результат выполнения такого запроса будет содержать 20×8 строк с большим количеством дублирующихся данных. Допустим, требуется найти только те книги, которые выпущены издательствами, перечисленными в таблице Publishers. Для обнаружения такого соответствия книг издательствам можно составить приведенный ниже запрос. Результат выполнения этого запроса

содержит 20 строк, т.е. по одной строке на каждую книгу, поскольку на каждую книгу в таблице Publishers приходится лишь одно издательство.

```
SELECT * FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

Если в запросе указано несколько таблиц, то в двух разных местах может упоминаться одно и то же имя столбца, как в показанном выше примере (столбец Publisher_Id из таблицы Books и аналогичный столбец Publisher_Id из таблицы Publishers). Во избежание неоднозначной интерпретации имен столбцов их следует предварять префиксом с именем таблицы, например Books.Publisher_Id.

Языковыми средствами SQL можно пользоваться и для изменения информации в базе данных. Допустим, требуется снизить на 5 долларов текущую цену всех книг, в названиях которых содержится подстрока "C++". С этой целью можно составить следующий запрос:

```
UPDATE Books
SET Price = Price - 5.00
WHERE Title LIKE '%C++%'
```

Аналогично для удаления всех книг по C++ понадобится оператор DELETE, как показано в приведенном ниже примере запроса. В языке SQL предусмотрены также встроенные функции для вычисления средних значений, поиска максимальных и минимальных значений в столбце и выполнения многих других действий, которые здесь не рассматриваются.

```
DELETE FROM Books
WHERE Title LIKE '%C++%'
```

Для ввода новых данных в таблицу обычно используется оператор INSERT:

```
INSERT INTO Books
VALUES ('A Guide to the SQL Standard', '0-201-96426-0',
        '0201', 47.95)
```

Для ввода каждой строки в таблицу приходится выполнять отдельный оператор INSERT. Но прежде чем составлять запросы, изменять и вводить данные, необходимо предоставить место для их хранения, т.е. создать таблицу. Для создания новой таблицы служит оператор CREATE TABLE, в котором указывается имя и тип данных каждого столбца, как показано в приведенном ниже примере.

```
CREATE TABLE Books
(
  Title CHAR(60),
  ISBN CHAR(13),
  Publisher_Id CHAR(6),
  Price DECIMAL(10,2)
)
```

В табл. 5.5 перечислены наиболее распространенные типы данных в SQL. Дополнительные предложения и операции, задающие ключи и ограничения, употребляемые в операторе CREATE TABLE, здесь не рассматриваются.

Таблица 5.5. Типы данных SQL

Тип данных	Описание
`INTEGER` или `INT`	Обычно 32-разрядное целое значение
`SMALLINT`	Обычно 16-разрядное целое значение
`NUMERIC(`*m*`, `*n*`)`, `DECIMAL(`*m*`, `*n*`)` или `DEC(`*m*`, `*n*`)`	Десятичное числовое значение с фиксированной точкой, содержащее *m* цифр, в том числе *n* знаков после точки
`FLOAT(`*n*`)`	Числовое значение с плавающей точкой и точностью до *n* знаков
`REAL`	Обычно 32-разрядное числовое значение с плавающей точкой
`DOUBLE`	Обычно 64-разрядное числовое значение с плавающей точкой
`CHARACTER(`*n*`)` или `CHAR(`*n*`)`	Строка фиксированной длины *n* символов
`VARCHAR(`*n*`)`	Строка переменной длины максимум *n* символов
`BOOLEAN`	Логическое значение
`DATE`	Календарная дата (зависит от реализации)
`TIME`	Время (зависит от реализации)
`TIMESTAMP`	Дата и время (зависят от реализации)
`BLOB`	Большой двоичный объект
`CLOB`	Большой символьный объект

5.3. Конфигурирование JDBC

Разумеется, для работы с базой данных потребуется система управления базой данных (СУБД), для которой в прикладном интерфейсе JDBC имеется подходящий драйвер. Среди имеющихся СУБД можно выбрать следующие: IBM DB2, Microsoft SQL Server, MySQL, Oracle или PostgreSQL.

Далее необходимо создать экспериментальную базу данных, например, под названием `COREJAVA`. Создайте новую базу данных сами или попросите сделать это администратора баз данных, а также наделить вас правами для создания, обновления и удаления таблиц.

Если вам не приходилось раньше устанавливать базу данных с архитектурой "клиент–сервер", то процесс ее установки, конечно, покажется вам очень сложным, а обнаружить причину возможной неудачи будет совсем не просто. Поэтому в таких случаях рекомендуется обратиться к услугам опытных специалистов.

Если у вас отсутствует опыт работы с базами данных, рекомендуется установить сначала базу данных Apache Derby, доступную для загрузки по адресу `http://db.apache.org/derby`. Прежде чем вы сможете написать свою первую программу для работы с базой данных, вам придется изучить ряд других вопросов, рассматриваемых в последующих разделах.

5.3.1. URL баз данных

Для подключения к базе данных необходимо указать ряд характерных для нее параметров. К их числу могут относиться имена хостов, номера портов, а также

имена баз данных. В прикладном интерфейсе JDBC используется синтаксис описания источника данных, подобный обычным URL. Ниже приведены некоторые примеры такого синтаксиса.

```
jdbc:derby://localhost:1527/COREJAVA;create=true
jdbc:postgresql:COREJAVA
```

Эти URL определяют в JDBC базы данных Derby и PostgreSQL по имени COREJAVA. Ниже приведена общая синтаксическая форма записи URL в JDBC, где *подчиненный_протокол* обозначает специальный драйвер для соединения с базой данных, а *другие_сведения* имеют формат, который зависит от применяемого подчиненного протокола. По поводу выбора конкретного формата следует обращаться к документации на применяемую базу данных.

```
jdbc:подчиненный_протокол:другие_сведения
```

5.3.2. Архивные JAR-файлы драйверов

Вам нужно будет также получить архивный JAR-файл, в котором находится драйвер для выбранной вами базы данных. Так, если вы пользуетесь базой данных Derby, вам понадобится файл derbyclient.jar. Если же это другая база данных, придется найти для нее подходящий драйвер. В частности, драйверы для базы данных PostgreSQL доступны для загрузки по адресу https://jdbc.postgresql.org.

При запуске программы, обращающейся к базе данных, в командной строке следует указать архивный JAR-файл драйвера после параметра -classpath. (Для компиляции самой программы архивный JAR-файл драйвера не требуется.) Для запуска подобных программ из командной строки можно воспользоваться приведенной ниже командой. В Windows текущий каталог, обозначаемый знаком ., отделяется от местонахождения архивного JAR-файла драйвера точкой с запятой.

```
java -classpath путь_к_файлу_драйвера:. имя_программы
```

5.3.3. Запуск базы данных

Прежде чем подключиться к серверу базы данных, его нужно запустить. Подробности этого процесса зависят от конкретной базы данных. В частности, для запуска базы данных Derby выполните следующие действия.

1. Откройте командную оболочку и перейдите в каталог, в котором будут находиться файлы базы данных.
2. Найдите архивный файл derbyrun.jar. В одних версиях JDK он может находиться в каталоге *jdk*/db/lib, а в других — в отдельном установочном каталоге JavaDB. Каталог, содержащий архивный файл lib/derbyrun.jar, обозначается здесь и далее как derby.
3. Выполните следующую команду:

   ```
   java -jar derby/lib/derbyrun.jar server start
   ```

4. Еще раз проверьте, работает ли база данных должным образом. Создайте файл ij.properties, введя в него следующие строки:

```
ij.driver=org.apache.derby.jdbc.ClientDriver
ij.protocol=jdbc:derby://localhost:1527/
ij.database=COREJAVA;create=true
```

5. Запустите в другой копии командной оболочки диалоговое инструментальное средство для написания сценариев базы данных Derby (оно называется `ij`), выполнив следующую команду:

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties
```

6. Теперь вы можете выдать команды SQL, например, следующие:

```
CREATE TABLE Greetings (Message CHAR(20));
INSERT INTO Greetings VALUES ('Hello, World!');
SELECT * FROM Greetings;
DROP TABLE Greetings;
```

7. Обратите внимание на то, что каждая команда должна завершаться точкой с запятой. Чтобы выйти из режима ввода команд SQL, введите команду `EXIT;`.

8. Завершив работу с базой данных, остановите ее сервер, выполнив следующую команду:

```
java -jar derby/lib/derbyrun.jar server shutdown
```

Если вы пользуетесь другой базой данных, вам нужно найти в документации на нее сведения о запуске и остановке сервера базы данных, а также о том, как подключаться к нему и выполнять команды SQL.

5.3.4. Регистрация класса драйвера

Многие архивные JAR-файлы прикладного интерфейса JDBC (например, драйвер базы данных Apache Derby) автоматически регистрируют класс драйвера. В этом случае вы можете пропустить этап ручной регистрации, рассматриваемый в этом разделе. Архивный JAR-файл может автоматически зарегистрировать класс драйвера, если он содержит файл `META-INF/services/java.sql.Driver`. Чтобы убедиться в этом, достаточно распаковать архивный JAR-файл драйвера.

Если же архивный JAR-файл драйвера не поддерживает автоматическую регистрацию, вам придется выяснить имена классов драйверов JDBC, используемых поставщиком вашей базы данных. Типичными именами классов драйверов являются следующие:

```
org.apache.derby.jdbc.ClientDriver
org.postgresql.Driver
```

Зарегистрировать драйвер с помощью класса `DriverManager` можно двумя способами. Первый состоит в том, чтобы загрузить класс драйвера в программу на Java, как показано в приведенной ниже строке кода, где выполняется статический инициализатор, который и осуществляет регистрацию загружаемого драйвера.

```
Class.forName("org.postgresql.Driver");
    // принудительно загрузить класс драйвера
```

Другой способ состоит в том, чтобы задать свойство `jdbc.drivers`, которое можно указать в качестве аргумента непосредственно в командной строке:

```
java -Djdbc.drivers=org.postgresql.Driver имя_программы
```

С другой стороны, вы можете установить системное свойство в своей прикладной программе, сделав следующий вызов:

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

При необходимости можно также указать несколько разных драйверов, разделив их двоеточием:

```
org.postgresql.Driver:org.apache.derby.jdbc.ClientDriver
```

5.3.5. Подключение к базе данных

Установить соединение с базой данных в прикладной программе на Java можно следующим образом:

```
String url = "jdbc:postgresql:COREJAVA";
String username = "dbuser";
String password = "secret";
Connection conn = DriverManager.getConnection(url,
                  username, password);
```

Диспетчер перебирает все зарегистрированные драйверы, пытаясь найти тот, который соответствует подчиненному протоколу, указанному в URL базы данных. Метод `getConnection()` возвращает объект типа `Connection`, который используется для выполнения операторов SQL. Чтобы соединиться с базой данных, необходимо знать имя пользователя базы данных и пароль.

НА ЗАМЕТКУ! По умолчанию база данных Derby допускает соединение под любым именем пользователя, не проверяя пароль. Для каждого пользователя в этой базе данных формируется отдельный ряд таблиц. По умолчанию используется имя пользователя **app**.

Все сказанное выше о работе с базами данных демонстрируется на примере тестовой программы, исходный код которой приведен в листинге 5.1. Эта программа загружает из файла свойств `database.properties` параметры подключения к базе данных и затем осуществляет его. Файл свойств `database.properties`, предоставляемый вместе с примером кода, содержит сведения о подключении к базе данных Derby. Если вы пользуетесь другой базой данных, введите этот в файл соответствующие сведения о подключении к конкретной базе данных. Ниже в качестве примера приведены параметры подключения к базе данных PostgreSQL.

```
jdbc.drivers=org.postgresql.Driver
jdbc.url=jdbc:postgresql:COREJAVA
jdbc.username=dbuser
jdbc.password=secret
```

После подключения к базе данных рассматриваемая здесь тестовая программа выполняет следующие операторы SQL:

```
CREATE TABLE Greetings (Message CHAR(20))
INSERT INTO Greetings VALUES ('Hello, World!')
SELECT * FROM Greetings
```

В результате выполнения оператора SELECT выводится приведенная ниже символьная строка.

```
Hello, World!
```

После этого таблица, созданная в базе данных, удаляется из нее в следующем операторе SQL:

```
DROP TABLE Greetings
```

Чтобы выполнить данную тестовую программу, запустите сначала базу данных, как описано выше, а затем саму программу, введя приведенную ниже команду. (Как всегда, пользователям Windows следует ввести точку с запятой (;) вместо двоеточия (:) для разделения составляющих пути к файлам.)

```
java -classpath .:driverJAR test.TestDB
```

СОВЕТ! Для устранения неполадок в прикладном интерфейсе JDBC можно активизировать трассировку JDBC. С этой целью следует вызвать метод **DriverManager.setLogWriter()**, чтобы направить сообщения трассировки в записывающий поток типа **PrintWriter**. Вывод трассировки содержит подробный перечень действий JDBC. В большинстве реализаций драйвера JDBC предоставляются дополнительные механизмы трассировки. Например, для базы данных Derby следует добавить параметр **traceFile** в URL прикладного интерфейса JDBC следующим образом:

```
jdbc:derby://localhost:1527/COREJAVA;create=true;traceFile=trace.out
```

Листинг 5.1. Исходный код из файла **test/TestDB.java**

```
1  package test;
2
3  import java.nio.file.*;
4  import java.sql.*;
5  import java.io.*;
6  import java.util.*;
7
8  /**
9   * В этой программе проверяется правильность
10  * конфигурирования базы данных и драйвера JDBC
11  * @version 1.03 2018-05-01
12  * @author Cay Horstmann
13  */
14 public class TestDB
15 {
16   public static void main(String args[])
17         throws IOException
18   {
19     try
20     {
21       runTest();
22     }
23     catch (SQLException ex)
24     {
25       for (Throwable t : ex)
26         t.printStackTrace();
```

```
27      }
28    }
29
30    /**
31     * Выполняет тест, создавая таблицу, вводя в
32     * нее значение, отображая содержимое таблицы
33     * и, наконец, удаляя ее
34     */
35    public static void runTest()
36          throws SQLException, IOException
37    {
38      try (Connection conn = getConnection();
39           Statement stat = conn.createStatement())
40      {
41        stat.executeUpdate("CREATE TABLE Greetings "
42                           + "(Message CHAR(20))");
43        stat.executeUpdate("INSERT INTO Greetings "
44                           + "VALUES ('Hello, World!')");
45
46        try (ResultSet result =
47             stat.executeQuery("SELECT * FROM Greetings"))
48        {
49          if (result.next())
50            System.out.println(result.getString(1));
51        }
52        stat.executeUpdate("DROP TABLE Greetings");
53      }
54    }
55
56    /**
57     * Получает сведения о подключении к базе данных
58     * из свойств, задаваемых в файле database.properties,
59     * и на их основании подключается к базе данных
60     * @return Подключение к базе данных
61     */
62    public static Connection getConnection()
63          throws SQLException, IOException
64    {
65      var props = new Properties();
66      try (InputStream in = Files.newInputStream(
67           Paths.get("database.properties")))
68      {
69        props.load(in);
70      }
71      String drivers = props.getProperty("jdbc.drivers");
72      if (drivers != null)
73        System.setProperty("jdbc.drivers", drivers);
74      String url = props.getProperty("jdbc.url");
75      String username = props.getProperty("jdbc.username");
76      String password = props.getProperty("jdbc.password");
77
78      return DriverManager.getConnection(url, username,password);
79    }
80  }
```

java.sql.DriverManager 1.1

- **static Connection getConnection(String *url*, String *user*, String *password*)**
  Устанавливает соединение с указанной базой данных и возвращает объект типа **Connection**.

5.4. Работа с операторами JDBC

В последующих разделах сначала будет показано, как пользоваться классом `Statement` из прикладного интерфейса JDBC для выполнения операторов SQL, получения результатов и обработки ошибок. Затем будет представлен пример простой программы для заполнения базы данных.

5.4.1. Выполнение операторов SQL

Для выполнения оператора SQL сначала создается объект типа `Statement`. Для этой цели используется объект типа `Connection`, который можно получить, вызвав метод `DriverManager.getConnection()` следующим образом:

```
Statement stat = conn.createStatement();
```

Затем формируется символьная строка с требующимся оператором SQL, как показано в приведенном ниже примере кода.

```
String command = "UPDATE Books"
      + " SET Price = Price - 5.00"
      + " WHERE Title NOT LIKE '%Introduction%'";
```

Далее вызывается метод `executeUpdate()` из класса `Statement`:

```
stat.executeUpdate(command);
```

Метод `executeUpdate()` возвращает количество строк, полученных из таблицы базы данных в результате выполнения оператора SQL, или же нуль строк для тех операторов, которые не возвращают количество строк из таблицы. Так, в результате приведенного выше вызова метода `executeUpdate()` возвращается количество книг, цена которых снижена на 5 долларов.

Вызывая метод `executeUpdate()`, можно выполнять операторы `INSERT`, `UPDATE` и `DELETE`, а также операторы определения данных, в том числе `CREATE TABLE` и `DROP TABLE`. Но для выполнения оператора `SELECT` необходимо вызвать другой метод — `executeQuery()`. Имеется также универсальный метод `execute()`, с помощью которого можно выполнять произвольные операторы SQL, но он применяется в основном для составления запросов в диалоговом режиме.

Если вы составляете запрос базы данных, вас, конечно, интересует результат его обработки. Метод `executeQuery()` возвращает объект типа `ResultSet`, который можно использовать для построчного просмотра результатов выполнения запроса:

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books")
```

Для анализа результирующего набора организуется приведенный ниже простой цикл.

```
while (rs.next())
{
  проанализировать строку из результирующего набора
}
```

ВНИМАНИЕ! Порядок последовательной обработки строк в интерфейсе **ResultSet** организован несколько иначе, чем в интерфейсе **java.util.Iterator**. В интерфейсе **ResultSet** итератор устанавливается на позиции *перед* первой строкой из результирующего набора. Поэтому для его перемещения к первой строке нужно вызвать метод **next()**. Кроме того, в данном интерфейсе отсутствует метод **hasNext()**, а следовательно, метод **next()** придется вызывать до тех пор, пока не будет возвращено логическое значение **false**.

Строки располагаются в результирующем наборе в совершено произвольном порядке. Если порядок их следования важен, его необходимо установить с помощью предложения `ORDER BY`. При обработке каждой строки таблицы обычно требуется получить содержимое отдельных полей (или столбцов). Для этой цели имеется целый ряд методов доступа к полям (или столбцам). Ниже приведены некоторые примеры вызова подобных методов доступа.

```
String isbn = rs.getString(1);
double price = rs.getDouble("Price");
```

Для каждого *типа данных* Java предусмотрен отдельный метод доступа, например `getString()` или `getDouble()`. И каждый из них реализован в двух формах: одной — с числовым параметром, а другой — со строковым. Если метод доступа вызывается с числовым параметром, данные извлекаются из столбца с указанным номером. Например, в результате вызова метода `rs.getString(1)` возвращается значение из первого столбца текущей строки таблицы.

ВНИМАНИЕ! В отличие от массивов, нумерация столбцов таблиц в базе данных начинается с **1**.

Если же метод доступа вызывается со строковым параметром, данные извлекаются из столбца с указанным именем. Например, в результате вызова метода `rs.getDouble("Price")` возвращается значение из столбца с именем `Price`. Первая форма методов доступа с числовым параметром более эффективна, но строковые параметры улучшают восприятие и упрощают сопровождение кода.

Если указанный тип не соответствует фактическому типу, метод доступа автоматически выполняет необходимое преобразование данных. Например, при вызове метода `rs.getString("Price")` числовое значение с плавающей точкой, извлекаемое из столбца `Price`, преобразуется в символьную строку.

java.sql.Connection 1.1

- **Statement createStatement()**
  Создает объект типа **Statement**, который может использоваться для выполнения операторов SQL без параметров.
- **void close()**
  Немедленно разрывает текущее соединение с базой данных и освобождает созданные для него ресурсы JDBC.

java.sql.Statement 1.1

- **ResultSet executeQuery(String *sqlQuery*)**

  Выполняет оператор SQL из указанной символьной строки и возвращает объект типа **ResultSet** с результатами выполнения этого оператора.

- **int executeUpdate(String *sqlStatement*)**
- **long executeLargeUpdate(String *sqlStatement*)** 8

  Выполняют операторы SQL типа **INSERT**, **UPDATE** и **DELETE** из указанной символьной строки, а также операторы языка определения данных (DDL) вроде **CREATE TABLE**. Возвращают количество строк, обработанных при выполнении данного оператора, или нулевое значение, если для данного оператора не установлен подсчет обновлений.

- **boolean execute(String *sqlStatement*)**

  Выполняет оператор SQL из указанной символьной строки и возвращает логическое значение **true**, если этот оператор предоставляет результирующий набор, а иначе — логическое значение **false**. Может сформировать множество результирующих наборов и подсчетов обновлений. Для доступа к данным, полученным в результате выполнения данного оператора SQL, следует вызвать метод **getResultSet()** или **getUpdateCount()**. Подробнее об обработке множественных результатов см. далее в разделе 5.5.4.

- **ResultSet getResultSet()**

  Возвращает объект типа **ResultSet** с результатами выполнения предыдущего оператора SQL или пустое значение **null**, если выполнение предыдущего оператора не дало никаких результатов. Этот метод следует вызывать только один раз для каждого выполняемого оператора SQL.

- **int getUpdateCount()**
- **long getLargeUpdateCount()** 8

  Возвращают количество строк, обработанных при выполнении предыдущего оператора обновления, или значение **-1**, если для данного оператора SQL не установлен подсчет обновлений. Эти методы следует вызывать только один раз для каждого выполняемого оператора SQL.

- **void close()**

  Закрывает данный оператор SQL и связанные с ним результирующие наборы.

- **boolean isClosed()** 6

  Возвращает логическое значение **true**, если данный оператор SQL закрыт.

- **void closeOnCompletion()** 7

  Закрывает данный оператор SQL, как только будут закрыты все связанные с ним результирующие наборы.

java.sql.ResultSet 1.1

- **boolean next()**

  Перемещает указатель текущей строки в результирующем наборе на одну позицию вперед. После прохождения последней строки возвращает логическое значение **false**. Следует, однако, иметь в виду, что данный метод должен быть вызван для перемещения указателя к первой строке в результирующем наборе.

java.sql.ResultSet 1.1 *(окончание)*

- ***Xxx*** **get*Xxx*(int *columnNumber*)**
- ***Xxx*** **get*Xxx*(String *columnLabel*)**
- (***Xxx*** обозначает тип данных, например **int**, **double**, **String**, **Date** и т.д.)
- **<T> T getObject(int *columnNumber*, Class<T> *type*)** 7
- **<T> T getObject(String *columnLabel*, Class<T> *type*)** 7

  Возвращает значение столбца, задаваемого номером или меткой, с преобразованием в указанный тип данных. Однако допускаются не все варианты преобразования типов. Метка столбца — это метка, указываемая в предложении **AS** оператора SQL, или имя столбца, если предложение **AS** не используется.
- **int findColumn(String *columnName*)**

  Возвращает номер столбца с указанным именем.
- **void close()**

  Немедленно закрывает текущий результирующий набор.
- **boolean isClosed()** 6

  Возвращает логическое значение **true** при закрытии данного оператора.

5.4.2. Управление подключениями, операторами и результирующими наборами

Каждый объект типа `Connection` может создать один или несколько объектов типа `Statement`. Один и тот же объект типа `Statement` можно использовать для выполнения нескольких не связанных между собой операторов SQL и запросов. Но для такого объекта допускается наличие *не более одного* открытого результирующего набора. Если же требуется выполнить несколько запросов и одновременно проанализировать их результаты, то для этого понадобится несколько объектов типа `Statement`.

Не следует, однако, забывать об ограничении, накладываемом на количество операторов SQL, приходящихся на одно подключение к базе данных. Количество открытых объектов типа `Statement`, одновременно поддерживаемых драйвером JDBC, можно выяснить, вызвав метод `getMaxStatements()` из интерфейса `DatabaseMetaData`.

На практике вряд ли стоит одновременно обрабатывать несколько результирующих наборов. Если же результирующие наборы взаимосвязаны, следует выдать составной запрос и проанализировать единый результат. Ведь намного эффективнее объединять запросы в самой базе данных, чем выполнять обход нескольких результирующих наборов в программе на Java.

Обработку любого результирующего набора следует завершить, прежде чем выдавать новый запрос или обновлять объект типа `Statement`. Ведь результирующие наборы из предыдущих запросов автоматически закрываются.

Покончив дело с объектом типа `ResultSet`, `Statement` или `Connection`, следует как можно скорее вызвать метод `close()`. Ведь эти объекты используют крупные структуры данных и истощаемые ресурсы на сервере базы данных. Метод `close()` из класса `Statement` автоматически закрывает результирующий набор, связанный с объектом данного класса, если, конечно, этот набор открыт

при выполнении соответствующей команды. Аналогично метод `close()` из класса `Connection` закрывает все объекты данного класса, открытые для соединения с базой данных.

С другой стороны, можно вызвать метод `closeOnCompletion()` из класса `Statement`, чтобы автоматически закрыть объект данного класса, как только будут закрыты все связанные с ним результирующие наборы. Если подключение к базе данных кратковременно, то можно и не беспокоиться о закрытии объектов типа `Statement` и связанных с ними результирующих наборов. Для абсолютной гарантии, что объект подключения к базе данных не останется открытым, можно воспользоваться оператором `try` с ресурсами следующим образом:

```
try (Connection conn = . . .)
{
  Statement stat = conn.createStatement();
  ResultSet result = stat.executeQuery(queryString);
  обработать результат запроса
}
```

5.4.3. Анализ исключений SQL

У каждого исключения SQL имеется цепочка объектов типа `SQLException`, которые извлекаются методом `getNextException()`. Эта цепочка исключений является дополнением "причинной" цепочки объектов типа `Throwable`, имеющихся в каждом исключении. (Подробнее об исключениях в Java см. в главе 7 первого тома данной книги.) Чтобы полностью перечислить все исключения, пришлось бы организовать два вложенных цикла. К счастью, класс `SQLException` был усовершенствован для реализации интерфейса `Iterable<Throwable>`. В частности, метод `iterator()` производит объект типа `Iterator<Throwable>`, который осуществляет перебор в обеих цепочках исключений, сначала проходя по "причинной" цепочке первого объекта типа `SQLException`, а затем переходя к следующему объекту типа `SQLException` и т.д. Для этой цели можно организовать усовершенствованный цикл `for` следующим образом:

```
for (Throwable t : sqlException)
{
  сделать что-нибудь с объектом t
}
```

Чтобы продолжить анализ объекта типа `SQLException`, для него можно вызвать методы `getSQLState()` и `getErrorCode()`. Первый метод выдает символьную строку по стандарту X/Open или SQL:2003. (Чтобы выяснить, какому именно стандарту соответствует применяемый драйвер, достаточно вызвать метод `getSQLStateType()` из интерфейса `DatabaseMetadata`.) Что же касается кода ошибки, то у различных поставщиков баз данных он разный.

Исключения SQL организованы в виде древовидной структуры наследования, приведенной на рис. 5.5. Благодаря этому имеется возможность перехватывать отдельные типы ошибок независимо от предпочтений поставщиков баз данных.

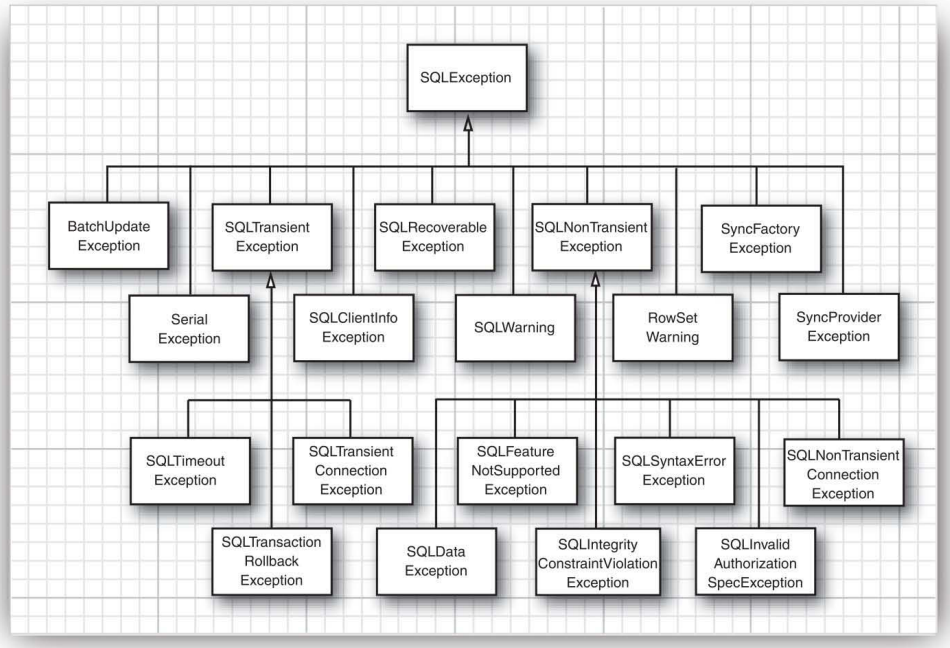


Рис. 5.5. Типы исключений SQL

Кроме того, драйвер базы данных может сообщать о некритичных ситуациях в виде предупреждений. Подобные предупреждения можно получать от подключений к базе данных, операторов и результирующих наборов. Класс `SQLWarning` является подклассом, производным от класса `SQLException`, несмотря на то, что объект типа `SQLWarning` не генерируется в виде исключения. Вызвав методы `getSQLState()` и `getErrorCode()`, можно получить дополнительные сведения о предупреждениях. Подобно исключениям SQL, предупреждения организуются в цепочку. И чтобы получить все предупреждения, нужно организовать следующий цикл:

```
SQLWarning w = stat.getWarning();
while (w != null)
{
  сделать что-нибудь с объектом w
  w = w.nextWarning();
}
```

Подкласс `DataTruncation`, производный от класса `SQLWarning`, используется в тех случаях, когда данные считываются из базы данных и в этот момент происходит их неожиданное усечение. Если усечение данных произошло при выполнении команды обновления, то объект типа `DataTruncation` генерируется в виде исключения.

java.sql.SQLException 1.1

- **SQLException getNextException()**
  Получает исключение SQL, следующее за данным исключением по цепочке, или пустое значение **null**, если достигнут конец цепочки.
- **Iterator<Throwable> iterator()** 6
  Получает итератор, перебирающий по цепочке исключения SQL и их причины.
- **String getSQLState()**
  Получает стандартный код ошибки, обозначающий состояние SQL.
- **int getErrorCode()**
  Получает код ошибки, характерный для поставщика используемой базы данных.

java.sql.SQLWarning 1.1

- **SQLWarning getNextWarning()**
  Получает предупреждение, следующее за данным исключением по цепочке, или пустое значение **null**, если достигнут конец цепочки.

java.sql.Connection 1.1
java.sql.Statement 1.1
java.sql.ResultSet 1.1

- **SQLWarning getWarnings()**
  Возвращает первое ожидающее предупреждение или пустое значение **null**, если ожидающие предупреждения отсутствуют.

java.sql.DataTruncation 1.1

- **boolean getParameter()**
  Возвращает логическое значение **true**, если усечение данных применяется к параметру, или логическое значение **false**, если оно применяется к столбцу.
- **int getIndex()**
  Возвращает индекс усеченного параметра или столбца.
- **int getDataSize()**
  Возвращает количество байтов, которые необходимо передать, или значение **-1**, если извлекаемое значение неизвестно.
- **int getTransferSize()**
  Возвращает количество байтов, которые были фактически переданы, или значение **-1**, если извлекаемое значение неизвестно.

5.4.4. Заполнение базы данных

Попробуем теперь написать первую реальную программу, используя прикладной интерфейс JDBC. Конечно, было бы неплохо, если бы в этой программе можно было выполнить некоторые из рассмотренных выше запросов. Но, к сожалению, это невозможно, потому что база данных пуста. Сначала ее нужно заполнить данными, хотя сделать это нетрудно с помощью ряда инструкций SQL для создания таблиц и ввода в них данных. Большинство СУБД способны обрабатывать инструкции SQL из текстового файла, но в этом случае проявляются досадные отличия в завершающих символах операторов и другие синтаксические особенности реализации SQL на разных платформах.

В силу этих причин воспользуемся прикладным интерфейсом JDBC, чтобы написать простую программу для построчного чтения инструкций SQL из файла и последующего их выполнения. В частности, рассматриваемая здесь программа должна читать данные из текстового файла в следующем формате:

```
CREATE TABLE Publishers (Publisher_Id CHAR(6),
                         Name CHAR(30), URL CHAR(80));
INSERT INTO Publishers VALUES ('0201', 'Addison-Wesley',
                               'www.aw-bc.com');
INSERT INTO Publishers VALUES ('0471', 'John Wiley & Sons',
                               'www.wiley.com');
. . .
```

В листинге 5.2 приведен исходный код программы `ExecSQL`, сначала читающей операторы SQL из текстового файла, а затем выполняющей их. Для применения этой программы совсем не обязательно разбираться в ее исходном коде. Самое главное, что она позволяет заполнить базу данных и выполнить примеры программ, приведенные в остальной части этой главы.

Прежде всего приведите сервер базы данных в рабочее состояние и запустите программу `ExecSQL` на выполнение, введя следующие команды:

```
java -classpath путь_к_драйверу:. exec.ExecSQL Books.sql
java -classpath путь_к_драйверу:. exec.ExecSQL Authors.sql
java -classpath путь_к_драйверу:. exec.ExecSQL Publishers.sql
java -classpath путь_к_драйверу:. exec.ExecSQL BooksAuthors.sql
```

Перед запуском данной программы проверьте содержимое файла свойств `database.properties` на соответствие вашей исполняющей среде (см. раздел 5.3.5 ранее в этой главе).

НА ЗАМЕТКУ! В состав вашей базы данных может входить утилита для непосредственного чтения файлов SQL. Например, в базе данных Derby можно выполнить приведенную ниже команду для чтения файла свойств **`ij.properties`**, описанного ранее в разделе 5.3.3.

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties Books.sql
```

В формате данных для программы **ExecSQL** допускается вставка точки с запятой в конце каждой строки, поскольку такой формат предполагается в большинстве утилит баз данных.

Ниже вкратце описываются основные этапы выполнения программы `ExecSQL`.

1. Устанавливается соединение с базой данных. Метод `getConnection()` считывает содержимое файла свойств `database.properties` и вводит свойство

jdbc.drivers в список системных свойств. Диспетчер драйверов использует свойство jdbc.drivers для загрузки соответствующего драйвера базы данных. Для подключения к базе данных в методе getConnection() применяются свойства jdbc.url, jdbc.username и jdbc.password.

2. Открывается текстовый файл с операторами SQL. Если такой файл отсутствует, пользователю предлагается ввести операторы вручную с консоли.
3. Все заданные операторы SQL выполняются с помощью универсального метода execute(). При получении результирующего набора этот метод возвращает логическое значение true. В конце всех четырех текстовых файлов с операторами SQL содержится оператор SELECT *. Это дает возможность убедиться, что информация успешно введена в базу данных.
4. При наличии результирующего набора полученные результаты выводятся на экран. А поскольку это обобщенный результирующий набор, то для определения количества столбцов в нем потребуются метаданные. Более подробно метаданные рассматриваются далее, в разделе 5.8.
5. Если при выполнении операторов SQL возникает какое-нибудь исключение, то выводятся сведения о нем, а также обо всех остальных исключениях, которые могут следовать по цепочке.
6. По завершении всех заданных операторов SQL соединение с базой данных разрывается.

Весь исходный код данной программы приведен в листинге 5.2.

Листинг 5.2. Исходный код из файла **exec/ExecSQL.java**

```
1  package exec;
2
3  import java.io.*;
4  import java.nio.charset.*;
5  import java.nio.file.*;
6  import java.util.*;
7  import java.sql.*;
8
9  /**
10  * Эта программа выполняет все команды SQL из
11  * текстового файла. Она вызывается следующим образом:
12  * java -classpath путь_к_драйверу:. ExecSQL
13  *       файл_с_операторами_SQL
14  * @version 1.33 2018-05-01
15  * @author Cay Horstmann
16  */
17 class ExecSQL
18 {
19   public static void main(String args[])
20         throws IOException
21   {
22     try (Scanner in = args.length == 0
23          ? new Scanner(System.in)
24          : new Scanner(Paths.get(args[0]),
25                        StandardCharsets.UTF_8))
```

```
26      {
27        try (Connection conn = getConnection();
28             Statement stat = conn.createStatement())
29        {
30          while (true)
31          {
32            if (args.length == 0)
33              System.out.println("Enter command or "
34                                 + "EXIT to exit:");
35            if (!in.hasNextLine()) return;
36
37            String line = in.nextLine().trim();
38            if (line.equalsIgnoreCase("EXIT")) return;
39            if (line.endsWith(";"))
40              // удалить точку с запятой в конце строки:
41              line = line.substring(0, line.length() - 1);
42            try
43            {
44              boolean isResult = stat.execute(line);
45              if (isResult)
46              {
47                try (ResultSet rs = stat.getResultSet())
48                {
49                  showResultSet(rs);
50                }
51              }
52              else
53              {
54                int updateCount = stat.getUpdateCount();
55                System.out.println(updateCount
56                                   + " rows updated");
57              }
58            }
59            catch (SQLException e)
60            {
61              for (Throwable t : e)
62                t.printStackTrace();
63            }
64          }
65        }
66        catch (SQLException e)
67        {
68          for (Throwable t : e)
69            t.printStackTrace();
70        }
71      }
72
73      /**
74       * Получает сведения о подключении к базе данных из
75       * свойств, задаваемых в файле database.properties,
76       * и на их основании подключается к базе данных
77       * @return Подключение к базе данных
78       */
79      public static Connection getConnection()
80            throws SQLException, IOException
81      {
82
```

```
83      var props = new Properties();
84      try (InputStream in = Files.newInputStream(
85           Paths.get("database.properties")))
86      {
87        props.load(in);
88      }
89      String drivers = props.getProperty("jdbc.drivers");
90      if (drivers != null)
91        System.setProperty("jdbc.drivers", drivers);
92
93      String url = props.getProperty("jdbc.url");
94      String username =
95             props.getProperty("jdbc.username");
96      String password =
97             props.getProperty("jdbc.password");
98
99      return DriverManager.getConnection(url,
100                         username, password);
101   }
102
103   /**
104    * Выводит результирующий набор
105    * @param result Выводимый результирующий набор
106    */
107   public static void showResultSet(ResultSet result)
108          throws SQLException
109   {
110     ResultSetMetaData metaData = result.getMetaData();
111     int columnCount = metaData.getColumnCount();
112
113     for (int i = 1; i <= columnCount; i++)
114     {
115       if (i > 1) System.out.print(", ");
116       System.out.print(metaData.getColumnLabel(i));
117     }
118     System.out.println();
119
120     while (result.next())
121     {
122       for (int i = 1; i <= columnCount; i++)
123       {
124         if (i > 1) System.out.print(", ");
125         System.out.print(result.getString(i));
126       }
127       System.out.println();
128     }
129   }
130 }
```

5.5. Выполнение запросов

В этом разделе рассматривается пример программы, способной выполнять запросы к базе данных COREJAVA. Для нормальной работы этой программы необходимо создать и заполнить таблицы в базе данных COREJAVA, как пояснялось в предыдущем разделе. При составлении запроса базы данных можно выбрать

автора книги и издательство или же оставить критерий отбора книги независимо от автора или издательства.

Рассматриваемая здесь программа позволяет также вносить изменения в содержимое базы данных. Для этого достаточно выбрать издательство и ввести сумму. Все цены на книги данного издательства автоматически откорректируются по введенной сумме, а программа отобразит количество измененных строк в таблице. После подобной коррекции цен на книги можно выполнить запрос, чтобы проверить новые цены.

5.5.1. Подготовленные операторы для запросов

В рассматриваемой здесь программе используется новое средство: *подготовленные операторы*. Рассмотрим следующий запрос SQL на выборку всех книг отдельного издательства независимо от их авторов:

```
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND Publishers.Name = название издательства,
                      выбираемое из списка
```

Вместо того чтобы создавать отдельный оператор SQL для каждого пользовательского запроса, можно заранее *подготовить* запрос с главной переменной и многократно использовать его, меняя только значение этой переменной. Такая возможность существенно повышает эффективность работы программы. Перед обработкой каждого запроса база данных вырабатывает план его эффективного исполнения. Подготавливая запрос для последующего многократного применения, можно исключить повторное планирование его выполнения.

Каждая главная переменная в запросе обозначается знаком вопроса (?). Если в запросе используется несколько главных переменных, необходимо внимательно следить за их расстановкой с помощью знаков вопроса, чтобы правильно устанавливать их конкретные значения. Ниже показано, как выглядит в исходном коде предварительно подготовленный запрос к рассматриваемой здесь базе данных COREJAVA.

```
String publisherQuery =
      "SELECT Books.Price, Books.Title "
    + "FROM Books, Publishers "
    + "WHERE Books.Publisher_Id = Publishers.Publisher_Id "
    + "AND Publishers.Name = ?";
PreparedStatement stat =
                  conn.prepareStatement(publisherQuery);
```

Перед выполнением подготовленного оператора необходимо связать главные переменные с их конкретными значениями, вызвав метод `set()`. Подобно разным формам метода `get()` из интерфейса `ResultSet`, для разных типов данных предусмотрены отдельные формы метода `set()`. В качестве примера ниже показано, каким образом задается строковое значение с названием издательства.

```
stat.setString(1, publisher);
```

Первый аргумент этого метода задает номер позиции главной переменной, обозначаемой знаком вопроса в подготовленном операторе, а второй аргумент — ее конкретное значение. Так, аргумент `1` задает первый знак `?`.

При повторном использовании подготовленного запроса с несколькими главными переменными все их привязки к конкретным значениям остаются в силе, если только они не изменены с помощью метода `set()` или `clearParameters()`. Это означает, что метод `setXxx()`, где *Xxx* — тип данных, следует вызывать только для тех главных переменных, которые изменяются в последующих запросах.

После привязки всех переменных к их конкретным значениям можно приступить к выполнению подготовленного оператора следующим образом:

```
ResultSet rs = stat.executeQuery();
```

СОВЕТ. Составление запроса вручную путем сцепления символьных строк — довольно трудоемкое, чреватое ошибками и небезопасное занятие. Ведь в этом случае нужно позаботиться об обозначении специальных символов (например, кавычек). А если при составлении запроса предполагается ввод пользователем данных, необходимо принять меры защиты от умышленного внесения запросов SQL при совершении атак на сервер базы данных. В этом отношении подготовленные операторы оказываются намного более удобными, и поэтому их рекомендуется применять всякий раз, когда в запрос включаются переменные.

Обновление цены осуществляется в операторе `UPDATE`. Обратите внимание на то, что для этого вызывается метод `executeUpdate()`, а не `executeQuery()`. Дело в том, что оператор `UPDATE` не возвращает результирующий набор, который в данном случае не требуется. Метод `executeUpdate()` возвращает лишь подсчет количества измененных строк в таблице, как показано ниже.

```
int r = stat.executeUpdate();
System.out.println(r + " rows updated");
```

НА ЗАМЕТКУ! Объект типа **`PreparedStatement`** становится недействительным после того, как связанный с ним объект типа **`Connection`** закрывается. Но многие драйверы баз данных автоматически *кешируют* подготовленные операторы. Если один и тот же запрос подготавливается дважды, то в базе данных еще раз используется план его выполнения. Поэтому, вызывая метод **`prepareStatement()`**, можно не особенно беспокоиться об издержках на выполнение подготовленных операторов.

Ниже вкратце описывается порядок действий, выполняемых в рассматриваемом здесь примере программы.

- Списочные массивы заполняются именами авторов и названиями издательств по двум запросам, из которых возвращаются все имена авторов и названия издательств, сохраняемые в базе данных.
- Запросы по имени автора имеют более сложную структуру. Ведь у одной книги может быть несколько авторов, и поэтому в таблице `BooksAuthors` сохраняется соответствие авторов и книг. Допустим, у книги с ISBN 0-201-96426-0 два автора с идентификаторами `DATE` и `DARW`. Для отражения этого факта таблица `BooksAuthors` должна содержать следующие строки:

  ```
  0-201-96426-0, DATE, 1
  0-201-96426-0, DARW, 2
  ```

- В третьем столбце указаны порядковые номера авторов. (Для этой цели нельзя использовать сведения о расположении строк в таблице, поскольку

в реляционной базе данных порядок следования записей не фиксирован.) Таким образом, в составляемом запросе следует сначала соединить таблицы Books, BooksAuthors и Authors, а затем сравнить в них имя автора с тем, что указано пользователем:

```
SELECT Books.Price, Books.Title FROM Books, BooksAuthors,
       Authors, Publishers
WHERE Authors.Author_Id = BooksAuthors.Author_Id
AND BooksAuthors.ISBN = Books.ISBN
AND Books.Publisher_Id = Publishers.Publisher_Id
AND Authors.Name = ?
AND Publishers.Name = ?
```

НА ЗАМЕТКУ! Некоторые программирующие на Java стараются избегать составления столь сложных запросов SQL. Как ни странно, они выбирают обходной, но неэффективный путь, предполагающий написание немалого объема кода на Java для последовательной обработки нескольких результирующих наборов. Следует, однако, иметь в виду, что база данных выполняет запросы *намного* эффективнее, чем программа на Java, поскольку база данных именно для этого и предназначена. В этой связи рекомендуется взять на вооружение следующее эмпирическое правило: то, что можно сделать средствами SQL, нецелесообразно делать средствами Java.

- Метод changePrices() выполняет оператор UPDATE. В предложении WHERE этой команды требуется указать *код* издательства, а известно лишь его *название*. Это затруднение разрешается с помощью вложенного запроса, как выделено ниже полужирным.

```
UPDATE Books
SET Price = Price + ?
WHERE Books.Publisher_Id =
      (SELECT Publisher_Id FROM Publishers WHERE Name = ?)
```

Весь исходный код данной программы приведен в листинге 5.3.

Листинг 5.3. Исходный код из файла **query/QueryTest.java**

```
1   package query;
2
3   import java.io.*;
4   import java.nio.file.*;
5   import java.sql.*;
6   import java.util.*;
7
8   /**
9    * В этой программе демонстрируется ряд сложных
10  * запросов базы данных
11  * @version 1.31 2018-05-01
12  * @author Cay Horstmann
13  */
14  public class QueryTest
15  {
16    private static final String allQuery =
17            "SELECT Books.Price, Books.Title FROM Books";
18
19    private static final String authorPublisherQuery =
20          "SELECT Books.Price, Books.Title "
```

```
21        + "FROM Books, BooksAuthors, Authors, Publishers "
22        + "WHERE Authors.Author_Id = "
23        + "BooksAuthors.Author_Id "
24        + "AND BooksAuthors.ISBN = Books.ISBN "
25        + "AND Books.Publisher_Id = "
26        + "Publishers.Publisher_Id AND Authors.Name = ? "
27        + "AND Publishers.Name = ?";
28
29     private static final String authorQuery =
30           "SELECT Books.Price, Books.Title FROM Books, "
31        + "BooksAuthors, Authors "
32        + "WHERE Authors.Author_Id =
33        + "BooksAuthors.Author_Id "
34        + "AND BooksAuthors.ISBN = Books.ISBN "
35        + " AND Authors.Name = ?";
36
37     private static final String publisherQuery =
38           "SELECT Books.Price, Books.Title FROM Books, "
39        + "Publishers "
40        + "WHERE Books.Publisher_Id = "
41        + "Publishers.Publisher_Id "
42        + "AND Publishers.Name = ?";
43
44     private static final String priceUpdate =
45           "UPDATE Books SET Price = Price + ? "
46        + " WHERE Books.Publisher_Id = "
47        + "(SELECT Publisher_Id "
48        + "FROM Publishers WHERE Name = ?)";
49
50     private static Scanner in;
51     private static ArrayList<String> authors =
52             new ArrayList<>();
53     private static ArrayList<String> publishers =
54             new ArrayList<>();
55
56     public static void main(String[] args)
57           throws IOException
58     {
59       try (Connection conn = getConnection())
60       {
61         in = new Scanner(System.in);
62         authors.add("Any");
63         publishers.add("Any");
64         try (Statement stat = conn.createStatement())
65         {
66           // заполнить списочный массив именами
67           // авторов книг
68           var query = "SELECT Name FROM Authors";
69           try (ResultSet rs = stat.executeQuery(query))
70           {
71             while (rs.next())
72               authors.add(rs.getString(1));
73           }
74
75           // заполнить списочный массив
76           // названиями издательств
77           query = "SELECT Name FROM Publishers";
```

```
78          try (ResultSet rs = stat.executeQuery(query))
79          {
80            while (rs.next())
81              publishers.add(rs.getString(1));
82          }
83        }
84        var done = false;
85        while (!done)
86        {
87          System.out.print(
88                "Q)uery C)hange prices E)xit: ");
89          String input = in.next().toUpperCase();
90          if (input.equals("Q"))
91            executeQuery(conn);
92          else if (input.equals("C"))
93            changePrices(conn);
94          else
95            done = true;
96        }
97      }
98      catch (SQLException e)
99      {
100       for (Throwable t : e)
101         System.out.println(t.getMessage());
102       }
103   }
104
105   /**
106    * Выполняет выбранный запрос
107    * @param conn Подключение к базе данных
108    */
109   private static void executeQuery(Connection conn)
110         throws SQLException
111   {
112     String author = select("Authors:", authors);
113     String publisher =
114           select("Publishers:", publishers);
115     PreparedStatement stat;
116     if (!author.equals("Any")
117        && !publisher.equals("Any"))
118     {
119       stat = conn.prepareStatement(
120             authorPublisherQuery);
121       stat.setString(1, author);
122       stat.setString(2, publisher);
123     }
124     else if (!author.equals("Any")
125           && publisher.equals("Any"))
126     {
127       stat = conn.prepareStatement(authorQuery);
128       stat.setString(1, author);
129     }
130     else if (author.equals("Any")
131           && !publisher.equals("Any"))
132     {
133       stat = conn.prepareStatement(publisherQuery);
134       stat.setString(1, publisher);
```

```
135     }
136     else
137       stat = conn.prepareStatement(allQuery);
138
139     try (ResultSet rs = stat.executeQuery())
140     {
141       while (rs.next())
142         System.out.println(rs.getString(1)
143                            + ", " + rs.getString(2));
144     }
145   }
146
147   /**
148    * Выполняет команду обновления с целью
149    * изменить цены на книги
150    * @param conn Подключение к базе данных
151    */
152   public static void changePrices(Connection conn)
153         throws SQLException
154   {
155     String publisher = select("Publishers:",
156           publishers.subList(1, publishers.size()));
157     System.out.print("Change prices by: ");
158     double priceChange = in.nextDouble();
159     PreparedStatement stat =
160           conn.prepareStatement(priceUpdate);
161     stat.setDouble(1, priceChange);
162     stat.setString(2, publisher);
163     int r = stat.executeUpdate();
164     System.out.println(r + " records updated.");
165   }
166
167   /**
168    * Предлагает пользователю выбрать символьную строку
169    * @param prompt Отображаемое приглашение
170    * @param options Варианты выбора,
171    *                предлагаемые пользователю
172    * @return Выбранный пользователем вариант
173    */
174   public static String select(String prompt,
175                               List<String> options)
176   {
177     while (true)
178     {
179       System.out.println(prompt);
180       for (int i = 0; i < options.size(); i++)
181         System.out.printf("%2d) %s%n", i + 1,
182                           options.get(i));
183       int sel = in.nextInt();
184       if (sel > 0 && sel <= options.size())
185         return options.get(sel - 1);
186     }
187   }
188
189   /**
190    * Получает сведения о подключении к базе данных из
191    * свойств, задаваемых в файле database.properties,
```

```
191     * и на их основании подключается к базе данных
192     * @return Подключение к базе данных
193     */
194    public static Connection getConnection()
195          throws SQLException, IOException
196    {
197      var props = new Properties();
198      try (InputStream in = Files.newInputStream(
199           Paths.get("database.properties")))
200      {
201        props.load(in);
202      }
203
204      String drivers = props.getProperty("jdbc.drivers");
205      if (drivers != null)
206        System.setProperty("jdbc.drivers", drivers);
207
208      String url = props.getProperty("jdbc.url");
209      String username =
210            props.getProperty("jdbc.username");
211      String password =
212            props.getProperty("jdbc.password");
213
214      return DriverManager.getConnection(url,
215                           username, password);
216    }
217 }
```

java.sql.Connection 1.1

- **PreparedStatement prepareStatement(String *sql*)**

  Возвращает объект типа **PreparedStatement**, содержащий подготовленный оператор. Заданная строка ***sql*** содержит оператор SQL с одной или несколькими заполнителями главных переменных, обозначенными вопросительными знаками.

java.sql.PreparedStatement 1.1

- **void set*Xxx*(int *n*, *Xxx x*)**
- (***Xxx*** обозначает тип данных, например **int**, **double**, **String**, **Date** и т.д.)

  Задает значение ***x*** для ***n***-го параметра.
- **void clearParameters()**

  Очищает все текущие параметры в подготовленном операторе.
- **ResultSet executeQuery()**

  Выполняет подготовленный запрос SQL и возвращает объект типа **ResultSet**.
- **int executeUpdate()**

  Выполняет операторы **INSERT**, **UPDATE** или **DELETE**, представленные в объекте типа **PreparedStatement** как подготовленные операторы SQL. Возвращает количество обработанных строк или нулевое значение для таких операторов языка DDL, как **CREATE TABLE**.

5.5.2. Чтение и запись больших объектов

Помимо чисел, символьных строк и дат, во многих базах данных можно сохранять *большие объекты* (LOB), к числу которых относятся изображения и другие данные. В языке SQL понятие больших объектов разделяется на категории больших двоичных объектов (BLOB) и больших символьных объектов (CLOB).

Чтобы прочитать большой объект, необходимо сначала выполнить команду `SELECT`, а затем вызвать метод `getBlob()` или `getClob()` из интерфейса `ResultSet`. В результате будет получен объект типа `Blob` или `Clob`. А для того чтобы получить двоичные данные из объекта типа `Blob`, следует вызвать метод `getBytes()` или `getInputStream()`. Так, если имеется таблица с изображениями на книжных обложках, то такое изображение можно получить следующим образом:

```
PreparedStatement stat = conn.prepareStatement(
        "SELECT Cover FROM BookCovers WHERE ISBN=?");
...
stat.set(1, isbn);
ResultSet result = stat.executeQuery();
if (result.next())
{
  Blob coverBlob = result.getBlob(1);
  Image coverImage = ImageIO.read(
                     coverBlob.getBinaryStream());
}
```

Аналогично, если извлечь объект типа `Clob`, то из него можно получить символьные данные, вызвав метод `String()` или `getCharacterStream()`.

Чтобы разместить большой объект в базе данных, следует вызвать метод `createLOB()` или `createClob()` для объекта типа `Connection`, получить поток вывода или поток записи для большого объекта, записать данные и сохранить этот объект в базе данных. В качестве примера ниже показано, как сохранить изображение в базе данных.

```
Blob coverBlob = connection.createBlob();
int offset = 0;
OutputStream out = coverBlob.setBinaryStream(offset);
ImageIO.write(coverImage, "PNG", out);
PreparedStatement stat = conn.prepareStatement(
        "INSERT INTO Cover VALUES (?, ?)");
stat.set(1, isbn);
stat.set(2, coverBlob);
stat.executeUpdate();
```

java.sql.ResultSet 1.1

- **Blob getBlob(int *columnIndex*)** 1.2
- **Blob getBlob(String *columnLabel*)** 1.2
- **Clob getClob(int *columnIndex*)** 1.2
- **Clob getClob(String *columnLabel*)** 1.2

  Получают большой двоичный объект (BLOB) или большой символьный объект (CLOB) из заданного столбца таблицы.

java.sql.Blob 1.2

- **long length()**
  Получает длину данного большого двоичного объекта.
- **byte[] getBytes(long *startPosition*, long *length*)**
  Получает данные в указанных пределах из текущего большого двоичного объекта.
- **InputStream getBinaryStream()**
- **InputStream getBinaryStream(long *startPosition*, long *length*)**
  Возвращают поток ввода для чтения данных из текущего большого двоичного объекта полностью или в указанных пределах.
- **OutputStream setBinaryStream(long *startPosition*)** 1.4
  Возвращает поток вывода для записи данных в текущий большой двоичный объект, начиная с указанной позиции.

java.sql.Clob 1.4

- **long length()**
  Получает количество символов в текущем большом символьном объекте.
- **String getSubString(long *startPosition*, long *length*)**
  Получает символы из текущего большого двоичного объекта в указанных пределах.
- **Reader getCharacterStream()**
- **Reader getCharacterStream(long *startPosition*, long *length*)**
  Возвращают поток чтения (а не поток ввода) символов из текущего большого символьного объекта в указанных пределах.
- **Writer setCharacterStream(long *startPosition*)** 1.4
  Возвращает поток записи (а не поток вывода) символов в текущий большой символьный объект, начиная с указанной позиции.

java.sql.Connection 1.1

- **Blob createBlob()** 6
- **Clob createClob()** 6
  Создают пустой большой двоичный объект (BLOB) или большой символьный объект (CLOB).

5.5.3. Синтаксис переходов в SQL

Синтаксис переходов предоставляет средства, которые обычно поддерживаются базами данных, но в разных вариантах в зависимости от конкретного синтаксиса базы данных. В задачу драйвера JDBC входит преобразование синтаксиса переходов в синтаксис конкретной базы данных.

Переходы предусмотрены для следующих средств.

- Литералы времени и даты.
- Вызовы скалярных функций.
- Вызовы хранимых процедур.
- Внешние соединения.
- Символы перехода в операциях LIKE.

Литералы даты и времени сильно отличаются в разных базах данных. Чтобы вставить литерал даты или времени, следует определить его значение в формате ISO 8601 (https://www.cl.cam.ac.uk/~mgk25/iso-time.html). После этого драйвер преобразует литерал в собственный формат базы данных. Для значений типа DATE, TIME или TIMESTAMP используются литералы d, t и ts следующим образом:

```
{d '2008-01-24'}
{t '23:59:59'}
{ts '2008-01-24 23:59:59.999'}
```

Скалярной называется такая функция, которая возвращает одно значение. В базах данных применяется немало скалярных функций, но под разными именами. В спецификации JDBC указаны стандартные имена, преобразуемые в имена, специфические для конкретных баз данных. Чтобы вызвать скалярную функцию, следует вставить ее стандартное имя и аргументы, как показано ниже. Полный список поддерживаемых имен скалярных функций можно найти в спецификации JDBC.

```
{fn left(?, 20)}
{fn user()}
```

Хранимой называется такая процедура, которая выполняется в базе данных и написана на специальном языке для конкретной базы данных. Для вызова хранимой процедуры служит переход call. Если у процедуры отсутствуют параметры, то указывать скобки не нужно. Для фиксации возвращаемого значения служит знак равенства. Ниже показано, каким образом вызываются хранимые процедуры.

```
{call PROC1(?, ?)}
{call PROC2}
{call ? = PROC3(?)}
```

Внешнее соединение двух таблиц не требует, чтобы строки из каждой таблицы совпадали по условию соединения. Например, в приведенном ниже запросе указаны книги, для которых столбец Publisher_Id не имеет совпадений в таблице Publishers, причем пустые значения NULL обозначают отсутствие совпадений.

```
SELECT * FROM {oj Books LEFT OUTER JOIN Publishers
ON Books.Publisher_Id = Publisher.Publisher_Id}
```

Чтобы включить в запрос издательства без совпадающих книг, может потребоваться предложение RIGHT OUTER JOIN, а чтобы возвратить по запросу и то и другое — предложение FULL OUTER JOIN. Синтаксис переходов требуется именно потому, что не во всех базах данных используется стандартное обозначение внешних соединений.

Наконец, знаки _ и % имеют специальное назначение в операции LIKE, обозначая совпадение с одним символом или последовательностью символов. Стандартного способа их буквального употребления не существует. Так, для сопоставления всех символьных строк, содержащих знак _, можно воспользоваться приведенной ниже конструкцией, где знак ! определен как символ перехода, а последовательность символов !_ буквально обозначает знак подчеркивания.

```
... WHERE ? LIKE %!_% {escape '!'}
```

5.5.4. Множественные результаты

По запросу могут быть возвращены множественные результаты. Это может произойти при выполнении хранимой процедуры или в базах данных, которые допускают также выполнение многих операторов SELECT в одном запросе. Получить все результирующие наборы можно следующим образом.

1. Вызвать метод execute() для выполнения оператора SQL.
2. Получить первый результат или подсчет обновлений.
3. Повторить вызов метода getMoreResults(), чтобы перейти к следующему результирующему набору.
4. Завершить процедуру, если больше не остается результирующих наборов или подсчетов обновлений.

Методы execute() и getMoreResults() возвращают логическое значение true, если следующим звеном в цепочке оказывается результирующий набор. Метод getUpdateCount() возвращает значение -1, если следующим звеном в цепочке *не* оказывается подсчет обновлений. В следующем цикле осуществляется последовательный обход всех полученных результатов:

```
boolean isResult = stat.execute(command);
boolean done = false;
while (!done)
{
  if (isResult)
  {
    ResultSet result = stat.getResultSet();
    сделать что-нибудь с полученным результатом
    в переменной result
  }
  else
  {
    int updateCount = stat.getUpdateCount();
    if (updateCount >= 0)
      сделать что-нибудь с подсчетом обновлений
      в переменной updateCount
    else
      done = true;
  }
  if (!done) isResult = stat.getMoreResults();
}
```

java.sql.Statement 1.1

- **boolean getMoreResults()**
- **boolean getMoreResults(int *current*)** 6

  Получают следующий результат выполнения данного оператора SQL. Параметр ***current*** принимает значение одной из следующих констант: **CLOSE_CURRENT_RESULT** (по умолчанию), **KEEP_CURRENT_RESULT** или **CLOSE_ALL_RESULTS**. Возвращает логическое значение **true**, если следующий результат существует и представляет собой результирующий набор.

5.5.5. Извлечение автоматически генерируемых ключей

В большинстве баз данных поддерживается механизм автоматической нумерации строк в таблице. К сожалению, у разных поставщиков баз данных эти механизмы заметно отличаются. Автоматически присваиваемые номера часто используются в качестве первичных ключей. Несмотря на то что в JDBC не предлагается независимое от особенностей разных баз данных решение для генерирования подобных ключей, в этом прикладном интерфейсе предоставляется эффективный способ их извлечения. Если при вводе новой строки в таблицу автоматически генерируется ключ, его можно получить с помощью следующего кода:

```
stmt.executeUpdate(insertStatement,
                   Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if (rs.next())
{
  int key = rs.getInt(1);
  . . .
}
```

java.sql.Statement 1.1

- **boolean execute(String *statement*, int *autogenerated*)** 1.4
- **int executeUpdate(String *statement*, int *autogenerated*)** 1.4

  Выполняют указанный оператор SQL, как пояснялось выше. Если параметр ***autogenerated*** принимает значение **Statement.RETURN_GENERATED_KEYS** и указан оператор **INSERT**, то первый столбец таблицы содержит автоматически сгенерированный ключ.

5.6. Прокручиваемые и обновляемые результирующие наборы

Как пояснялось ранее, метод `next()` из интерфейса `ResultSet` позволяет последовательно перебирать строки в результирующем наборе, получаемом по запросу базы данных. Его очень удобно использовать для анализа полученных данных. Но нередко пользователю требуется предоставить возможность для просмотра результатов выполнения запроса с переходом к предыдущей и следующей строке, как было, например, показано на рис. 5.4. В *прокручиваемом* результирующем наборе можно свободно перемещаться не только к предыдущим и последующим записям, но и на произвольную позицию.

Кроме того, при просмотре результатов выполнения запроса у пользователей часто возникает потребность исправить какие-нибудь данные. В *обновляемом* результирующем наборе можно видоизменять записи программно, чтобы автоматически обновить их в базе данных. Все эти возможности обработки результирующих наборов обсуждаются в последующих разделах.

5.6.1. Прокручиваемые результирующие наборы

По умолчанию результирующие наборы не являются прокручиваемыми или обновляемыми. Для организации прокрутки результатов выполнения запроса необходимо получить объект типа `Statement` следующим образом:

```
Statement stat = conn.createStatement(type, concurrency);
```

Для подготовленного оператора потребуется следующий вызов:

```
PreparedStatement stat = conn.prepareStatement(
                         command, type, concurrency);
```

Допустимые значения параметров `type` и `concurrency` перечислены в табл. 5.6 и 5.7. Выбирая эти значения, придется найти ответы на следующие вопросы.

- Требуется ли сделать результирующий набор прокручиваемым? Если этого не требуется, следует выбрать значение `ResultSet.TYPE_FORWARD_ONLY`.
- Если все же требуется сделать результирующий набор прокручиваемым, то должен ли он отражать те данные, которые были изменены в базе данных после выполнения запроса? (Здесь и далее предполагается, что установлен параметр `ResultSet.TYPE_SCROLL_INSENSITIVE`, т.е. результирующий набор не реагирует на те изменения, которые произошли в базе данных после выполнения запроса.)
- Требуется ли отредактировать результирующий набор и обновить базу данных? (Более подробно этот вопрос рассматривается в следующем разделе.)

Таблица 5.6. Значения параметра **`type`**, представленные константами из интерфейса **`ResultSet`**

Значение	Описание
`TYPE_FORWARD_ONLY`	Без прокрутки (по умолчанию)
`TYPE_SCROLL_INSENSITIVE`	С прокруткой, но без учета изменений в базе данных
`TYPE_SCROLL_SENSITIVE`	С прокруткой и с учетом изменений в базе данных

Таблица 5.7. Значения параметра **`concurrency`**, представленные константами из интерфейса **`ResultSet`**

Значение	Описание
`CONCUR_READ_ONLY`	Без редактирования и обновления базы данных (по умолчанию)
`CONCUR_UPDATABLE`	С редактированием и обновлением базы данных

Так, если требуется только прокрутка результирующего набора, но не редактирование его данных, это можно организовать следующим образом:

```
Statement stat = conn.createStatement(
                 ResultSet.TYPE_SCROLL_INSENSITIVE,
                 ResultSet.CONCUR_READ_ONLY);
```

Теперь можно прокручивать все результирующие наборы, возвращаемые при вызовах приведенного ниже метода. Получаемый в итоге результирующий набор содержит *курсор*, устанавливаемый на текущей позиции.

```
ResultSet rs = stat.executeQuery(query)
```

НА ЗАМЕТКУ! Не все драйверы баз данных поддерживают прокручиваемые или обновляемые результирующие наборы. (Методы **supportsResultSetType()** и **supportsResultSetConcurrency()** из интерфейса **DatabaseMetaData** сообщают о типах и режимах параллельной обработки, которые поддерживаются в конкретной базе данных с помощью определенного драйвера.) Но даже если в базе данных поддерживаются результирующие наборы во всех описанных режимах, то в некоторых запросах нельзя получить результирующий набор со всеми запрашиваемыми свойствами. (Например, результат выполнения сложного запроса может оказаться необновляемым.)

В этом случае метод **executeQuery()** возвращает результирующий набор типа **ResultSet** с меньшими возможностями, вводя предупреждение типа **SQLWarning** в объект соединения. (Способ извлечения предупреждений представлен ранее, в разделе 5.4.3.) С другой стороны, для выявления конкретного режима работы результирующего набора можно вызвать методы **getType()** и **getConcurrency()** из интерфейса **ResultSet**. Если не выяснить конкретный режим работы результирующего набора и попытаться выполнить неподдерживаемую в нем операцию, например, вызвать метод **previous()** для непрокручиваемого результирующего набора, это неизбежно приведет к исключению типа **SQLException**.

Прокрутка организуется очень просто. Например, для перехода к предыдущим записям в результирующем наборе служит приведенная ниже конструкция. Метод `previous()` возвращает логическое значение `true`, если курсор находится на конкретной строке в результирующем наборе, или логическое значение `false`, если курсор находится перед первой строкой.

```
if (rs.previous()) . . .
```

Для перемещения курсора на *n* строк вперед или назад вызывается следующий метод:

```
rs.relative(n);
```

При положительных значениях параметра *n* курсор перемещается вперед, а при отрицательных — назад (нулевое значение параметра *n* не приводит ни к каким перемещениям). Если попытаться переместить курсор за пределы текущего ряда строк, он расположится за последней строкой или же перед первой строкой в зависимости от знака в значении параметра *n*. После этого метод `relative()` возвращает логическое значение `false`, а перемещение курсора прекращается. Данный метод возвращает логическое значение `true` только в том случае, если курсор устанавливается на конкретной строке.

С другой стороны, курсор можно установить на конкретной строке под номером *n*, вызвав следующий метод:

```
s.absolute(n);
```

Получить текущий номер строки *n* можно следующим образом:

```
int currentRow = rs.getRow();
```

Первая строка в результирующем наборе имеет номер 1. Если возвращаемое значение равно нулю, то курсор находится не на конкретной строке, а за последней или перед первой строкой. Для установки курсора на первой или последней строке, перед первой или за последней строкой результирующего набора предусмотрены удобные методы `first()`, `last()`, `beforeFirst()` и `afterLast()` соответственно, а для проверки расположения курсора на одной из этих позиций — удобные методы `isFirst()`, `isLast()`, `isBeforeFirst()` и `isAfterLast()`.

Как видите, обрабатывать прокручиваемые результирующие наборы совсем не трудно. Все рутинные операции, связанные с кешированием данных, получаемых по запросу, выполняются драйвером базы данных.

5.6.2. Обновляемые результирующие наборы

Если требуется отредактировать данные, полученные по запросу в результирующем наборе, а также автоматически обновить базу данных, такой набор следует сделать обновляемым. Обновляемые результирующие наборы совсем не обязательно должны быть прокручиваемыми. Но если требуется предоставить пользователю возможность редактировать данные, то они должны допускать и прокрутку.

Для получения обновляемого результирующего набора служит приведенный ниже код, где для этой цели в качестве параметра указана константа, выделенная полужирным. Результирующие наборы, возвращаемые методом `executeQuery()`, становятся в итоге обновляемыми.

```
Statement stat = conn.createStatement(
                 ResultSet.TYPE_SCROLL_INSENSITIVE,
                 ResultSet.CONCUR_UPDATABLE);
```

НА ЗАМЕТКУ! Обновляемый результирующий набор возвращается не по всем запросам. Так, если в запросе предполагается соединение нескольких таблиц, его результат не всегда может быть обновляемым. Но если в запросе предполагается обращение к одной таблице или соединение нескольких таблиц по их первичным ключам, то следует ожидать, что получаемый в итоге результирующий набор окажется обновляемым. Чтобы выяснить, является ли результирующий набор обновляемым, следует вызвать метод **`getConcurrency()`** из интерфейса **`ResultSet`**.

Допустим, требуется повысить цену на некоторые книги, но отсутствует единый критерий, который можно было бы использовать для этого в команде `UPDATE`. В таком случае придется перебрать в цикле все книги и изменить цены по произвольным условиям, как показано ниже.

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next())
{
  if (. . .)
  {
    double increase = . . .
    double price = rs.getDouble("Price");
    rs.updateDouble("Price", price + increase);
    rs.updateRow(); // непременно вызвать метод updateRow()
                    // после обновления полей в таблице
  }
}
```

Для всех типов данных SQL предусмотрены соответствующие формы метода `updateXxx()`, например `updateDouble()`, `updateString()` и т.д. Как и при вызове различных форм метода `getXxx()`, в качестве параметров данного метода могут быть указаны номер или имя столбца, а затем новое значение поля.

НА ЗАМЕТКУ! Применяя различные формы метода **`updateXxx()`**, следует иметь в виду, что первый его параметр обозначает номер столбца в результирующем наборе, где он может отличаться от номера столбца в базе данных.

В методе `updateXxx()` изменяется только значения полей в текущей строке результирующего набора, а не в самой базе данных. Для обновления всех полей отредактированной строки в базе данных следует вызвать метод `updateRow()`. Если же переместить курсор к следующей строке, не вызывая метод `updateRow()`, все обновления предыдущей строки в результирующем наборе будут отменены, поскольку они не были переданы базе данных. Для отмены обновлений текущей строки в базе данных следует вызвать метод `cancelRowUpdates()`.

В предыдущем примере был продемонстрирован порядок внесения изменений в существующей строке. Для создания новой строки в базе данных нужно сначала вызвать метод `moveToInsertRow()`, переместив тем самым курсор на специальную позицию, называемую *строкой вставки*. Затем новая строка создается на данной позиции с помощью метода `updateXxx()`. А для передачи новой вставляемой строки в базу данных вызывается метод `insertRow()`. По окончании вставки вызывается метод `moveToCurrentRow()`, чтобы переместить курсор назад на ту позицию, которую он занимал до вызова метода `moveToInsertRow()`. В приведенном ниже примере показано, каким образом весь этот процесс реализуется непосредственно в коде.

```
rs.moveToInsertRow();
rs.updateString("Title", title);
rs.updateString("ISBN", isbn);
rs.updateString("Publisher_Id", pubid);
rs.updateDouble("Price", price);
rs.insertRow();
rs.moveToCurrentRow();
```

Однако конкретное *расположение* новых данных в результирующем наборе или базе данных *не* поддается непосредственному управлению из прикладного кода. Если не указать конкретное значение для столбца в строке вставки, на этом месте окажется пустое значение `NULL`. Но если на столбец наложено ограничение `NOT NULL`, то сгенерируется исключение и строка не будет вставлена.

Наконец, для удаления той строки, на которой установлен курсор, вызывается следующий метод, который немедленно удаляет строку как из результирующего набора, так из базы данных:

```
rs.deleteRow();
```

Таким образом, методы `updateRow()`, `insertRow()` и `deleteRow()` из интерфейса `ResultSet` предоставляют те же возможности, что и операторы `UPDATE`, `INSERT` и `DELETE` языка SQL. Для программирующих на Java вызов методов более привычен, поэтому они предпочитают данный подход составлению запросов из операторов SQL.

ВНИМАНИЕ! При неаккуратной обработке обновляемых результирующих наборов можно получить совершенно неэффективный код. Нередко выполнение оператора **UPDATE** оказывается *намного* более эффективным, чем составление запроса и просмотр результирующего набора. Обработку обновляемых результирующих наборов имеет смысл организовывать в диалоговых прикладных программах, где пользователь может вносить произвольные изменения. Для внесения заранее программируемых изменений более подходящим оказывается оператор **UPDATE**.

НА ЗАМЕТКУ! В версии JDBC 2 были внедрены дополнительные усовершенствования в результирующие наборы, в том числе возможность обновлять результирующие наборы самыми последними данными, если они были изменены при другом, параллельном соединении с базой данных. В версии JDBC 3 было внедрено еще одно усовершенствование, определяющее режим работы результирующих наборов при фиксации транзакции. Но эти дополнительные возможности здесь не рассматриваются, поскольку они выходят за рамки введения в базы данных. За дополнительными сведениями о них отсылаем читателей к книге *JDBC™ API Tutorial and Reference, Third Edition* Мэйдена Фишера, Джона Эллиса и Джонатана Брюса (Maydene Fisher, Jon Ellis, Jonathan Bruce; издательство Addison-Wesley, 2003 г.), а также к документации, описывающей спецификацию прикладного интерфейса JDBC и доступной для загрузки по адресу **https://download.oracle.com/otndocs/jcp/jdbc-4_2-mrel2-spec/**.

java.sql.Connection 1.1

- **Statement createStatement(int *type*, int *concurrency*)** 1.2
- **PreparedStatement prepareStatement(String *command*, int type, int *concurrency*) 1.2**

  Создают обычный или подготовленный оператор SQL и возвращают результирующий набор по заданному типу и способу параллельного обращения к нему. Параметр **type** принимает одну из констант **TYPE_FORWARD_ONLY**, **TYPE_SCROLL_INSENSITIVE** или **TYPE_SCROLL_SENSITIVE**, а параметр ***concurrency*** — константу **CONCUR_READ_ONLY** или **CONCUR_UPDATABLE**. Все перечисленные константы определяются в интерфейсе **ResultSet**.

java.sql.ResultSet 1.1

- **int getType()** 1.2

  Возвращает одну из констант (**TYPE_FORWARD_ONLY**, **CONCUR_UPDATABLE** или **TYPE_SCROLL_SENSITIVE**), обозначающих тип результирующего набора.
- **int getConcurrency()** 1.2

  Возвращает константу (**CONCUR_READ_ONLY** или **CONCUR_UPDATABLE**), обозначающую способ параллельного обращения к результирующему набору (только для чтения или обновления).
- **boolean previous()** 1.2

  Перемещает курсор к предыдущей строке. Возвращает логическое значение **true**, если курсор устанавливается на строке, или логическое значение **false**, если он устанавливается перед первой строкой.

java.sql.ResultSet 1.1 *(окончание)*

- **int getRow()** 1.2

  Получает номер текущей строки. Нумерация строк начинается с **1**.

- **boolean absolute(int *r*)** 1.2

  Перемещает курсор к строке с номером ***r***. Возвращает логическое значение **true**, если курсор устанавливается на строке.

- **boolean relative(int *d*)** 1.2

  Перемещает курсор на количество строк, определяемое параметром ***d***. Если значение параметра ***d*** меньше нуля, то перемещение происходит в обратном направлении. Возвращает логическое значение **true**, если курсор устанавливается на строке.

- **boolean first()** 1.2
- **boolean last()** 1.2

  Перемещают курсор к первой или к последней строке. Возвращают логическое значение **true**, если курсор устанавливается на строке.

- **void beforeFirst()** 1.2
- **void afterLast()** 1.2

  Устанавливают курсор перед первой или за последней строкой.

- **boolean isFirst()** 1.2
- **boolean isLast()** 1.2

  Проверяют, находится ли курсор на первой или на последней строке.

- **boolean isBeforeFirst()** 1.2
- **boolean isAfterLast()** 1.2

  Проверяют, находится ли курсор перед первой или за последней строкой.

- **void moveToInsertRow()** 1.2

  Перемещает курсор на строку вставки. Строка вставки — это специальная строка, которая служит для вставки новых данных с помощью методов **update*Xxx*()** и **innsertRow()**.

- **void moveToCurrentRow()** 1.2

  Перемещает курсор из строки вставки на строку, где он находился до вызова метода **moveToInsertRow()**.

- **void insertRow()** 1.2

  Вводит содержимое строки вставки в базу данных и результирующий набор.

- **void deleteRow()** 1.2

  Удаляет текущую строку из базы данных и результирующего набора.

- **void update*Xxx*(int *column*, *Xxx data*)** 1.2
- **void update*Xxx*(String *columnName*, *Xxx data*)** 1.2

  [***Xxx*** обозначает тип данных, например **int**, **double**, **String**, **Date** и т.д.]

  Обновляют содержимое указанного столбца из текущей строки в результирующем наборе.

- **void updateRow()** 1.2

  Передает обновления текущей строки в базу данных.

- **void cancelRowUpdates ()** 1.2

  Отменяет обновления текущей строки.

java.sql.DatabaseMetaData 1.1

- **boolean supportsResultSetType(int *type*)** 1.2

  Возвращает логическое значение **true**, если база данных способна поддерживать заданный тип результирующего набора. Параметр **type** принимает одну из констант **TYPE_FORWARD_ONLY**, **TYPE_SCROLL_INSENSITIVE** или **TYPE_SCROLL_SENSITIVE**, определяемых в интерфейсе **ResultSet**.

- **boolean supportsResultSetConcurrency(int *type*, int *concurrency*)** 1.2

  Возвращает логическое значение **true**, если база данных способна поддерживать заданный способ прокрутки и параллельного обращения к результирующему набору. Параметр **type** принимает одну из констант **TYPE_FORWARD_ONLY**, **TYPE_SCROLL_INSENSITIVE** или **TYPE_SCROLL_SENSITIVE**, а параметр ***concurrency*** — константу **CONCUR_READ_ONLY** или **CONCUR_UPDATABLE**. Все перечисленные константы определяются в интерфейсе **ResultSet**.

5.7. Наборы строк

Прокручиваемые результирующие наборы предлагают богатые возможности, но они не свободны от недостатков. В течение всего периода взаимодействия с пользователем должно быть установлено соединение с базой данных. Но ведь пользователь может отлучиться на длительное время, а между тем установленное соединение будет напрасно потреблять сетевые ресурсы. В подобной ситуации целесообразно использовать *набор строк*. Интерфейс `RowSet` расширяет интерфейс `ResultSet`, но набор строк не должен быть привязан к соединению с базой данных.

Наборы строк применяются и в том случае, если требуется перенести результаты выполнения запроса на другой уровень сложного приложения или на другое устройство, например на мобильный телефон. Перенести результирующий набор нельзя, поскольку он связан с подключением к базе данных, а кроме того, структура данных может иметь довольно крупные размеры.

5.7.1. Построение наборов строк

Ниже перечислены интерфейсы, входящие в пакет `javax.sql.rowset` и расширяющие интерфейс `RowSet`.

- Интерфейс `CachedRowSet` позволяет выполнять некоторые операции при отсутствии соединения. Кешируемые наборы строк рассматриваются в следующем разделе.
- Интерфейс `WebRowSet` представляет кешируемый набор строк, который может быть сохранен в XML-файле. Сам XML-файл может быть передан другому компоненту приложения и открыт с помощью другого объекта типа `WebRowSet`.
- Интерфейсы `FilteredRowSet` и `JoinRowSet` поддерживают легковесные операции с наборами строк, равнозначные таким командам SQL, как `SELECT` и `JOIN`. Эти операции выполняются только над данными,

содержащимися в наборе строк, для чего не требуется устанавливать соединение с базой данных.

- Интерфейс `JdbcRowSet` является тонкой оболочкой для интерфейса `ResultSet`, вводя удобные методы из интерфейса `RowSet`.

В версии Java 7 появился стандартный способ получения набора строк с помощью приведенных ниже методов. Аналогичные методы имеются для получения наборов строк других типов.

```
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
```

5.7.2. Кешируемые наборы строк

Кешируемый набор строк содержит данные из результирующего набора. Интерфейс `CachedRowSet` расширяет интерфейс `ResultSet`, а следовательно, кешируемым набором строк можно пользоваться точно так же, как и результирующим. Но наборы строк имеют существенное преимущество, позволяющее отключиться от базы данных и продолжать работать с набором строк. Как демонстрируется в примере программы, исходный код которой представлен в листинге 5.4, такая возможность существенно упрощает создание диалоговых приложений. При получении команды от пользователя осуществляется подключение к базе данных, выполняется запрос, результаты размещаются в наборе строк, после чего производится отключение от базы данных.

Кешируемый набор строк позволяет даже видоизменить содержащиеся в нем данные. Разумеется, результаты подобных изменений не отражаются в базе данных немедленно. Чтобы принять накопленные изменения, необходимо выполнить явный запрос. В этом случае кешируемый набор строк типа `CachedRowSet` повторно подключается к базе данных и выполняет операторы SQL для записи изменений в базу данных. Кешируемый набор строк типа `CachedRowSet` заполняется данными из результирующего набора следующим образом:

```
ResultSet result = . . .;
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
crs.populate(result);
conn.close(); // теперь можно отключиться от базы данных
```

С другой стороны, кешируемому набору строк типа `CachedRowSet` можно предоставить возможность автоматически подключиться к базе данных. Для этого сначала задаются следующие параметры базы данных:

```
crs.setURL("jdbc:derby://localhost:1527/COREJAVA");
crs.setUsername("dbuser");
crs.setPassword("secret");
```

Затем составляется оператор запроса с любыми параметрами, как показано ниже.

```
crs.setCommand("SELECT * FROM Books WHERE PUBLISHER = ?");
crs.setString(1, publisherName);
```

Наконец, набор строк заполняется результатами запроса. В результате приведенного ниже вызова осуществляется подключение к базе данных, выполняется запрос, заполняется набор строк и производится отключение от базы данных.

```
crs.execute();
```

Если полученный результат запроса имеет слишком большой объем, его можно и не полностью вводить в набор строк. В конце концов, пользователи прикладной программы, скорее всего, просмотрят лишь некоторые строки. В таком случае придется определить размеры страницы следующим образом:

```
CachedRowSet crs = . . .;
crs.setCommand(command);
crs.setPageSize(20);
. . .
crs.execute();
```

В итоге будет доступно только 20 строк. Для того чтобы получить следующую порцию строк, достаточно вызвать метод

```
crs.nextPage();
```

Для просмотра и видоизменения набора строк служат те же методы, что и для обращения с результирующим набором. Так, если изменить содержимое набора строк, для записи изменений в базу данных необходимо сделать один из следующих вызовов:

```
crs.acceptChanges(conn);
```

или

```
crs.acceptChanges();
```

Второй вариант вызова метода `acceptChanges()` действует только в том случае, если предоставить для набора строк все сведения, необходимые для подключения к базе данных (URL, имя пользователя и пароль).

Как упоминалось в разделе 5.6.2, не все результирующие наборы являются обновляемыми. Аналогично наборы строк, содержащие результаты сложных запросов, не позволяют записывать изменения в базу данных. Если же набор строк содержит данные только из одной таблицы, то никаких затруднений при их записи в базу данных не возникает.

ВНИМАНИЕ! Если заполнить набор строк данными из результирующего набора, то набору строк не будет известно имя обновляемой таблицы. В таком случае придется специально указать имя таблицы, вызвав метод **`setTableName()`**.

Если данные в базе изменились с того момента, как набор строк был заполнен ими, то возникают дополнительные затруднения, связанные с несоответствием данных. В базовой реализации проверяется, совпадают ли исходные значения из набора строк (т.е. значения перед редактированием) с текущими значениями в базе данных. Если проверка дает положительный результат, то содержимое базы данных заменяется видоизмененными данными. В противном случае генерируется исключение типа `SyncProviderException` и внесенные изменения не записываются. В других реализациях могут применяться иные способы синхронизации данных.

javax.sql.RowSet 1.4

- **String getURL()**
- **void setURL(String *url*)**

  Получают или устанавливают URL базы данных.
- **String getUsername()**
- **void setUsername(String *username*)**

  Получают или устанавливают имя пользователя для подключения к базе данных.
- **String getPassword()**
- **void setPassword(String *password*)**

  Получают или устанавливают пароль для подключения к базе данных.
- **String getCommand()**
- **void setCommand(String *command*)**

  Получают или устанавливают команду, при выполнении которой набор строк заполняется данными.
- **void execute()**

  Заполняет данный набор строк по команде, установленной с помощью метода **setCommand()**. Для того чтобы диспетчер драйверов мог подключиться к базе данных, должны быть заданы URL, имя пользователя и пароль.

javax.sql.rowset.CachedRowSet 5.0

- **void execute(Connection *conn*)**

  Заполняет набор строк по команде, установленной с помощью метода **setCommand()**. Использует указанное подключение к базе данных, а затем *отключается* от нее.
- **void populate(ResultSet *result*)**

  Заполняет кешируемый набор строк данными из указанного результирующего набора.
- **String getTableName()**
- **void setTableName(String *tableName*)**

  Получают или устанавливают имя таблицы, данными из которой заполняется кешируемый набор строк.
- **int getPageSize()**
- **void setPageSize(int *size*)**

  Получают или устанавливают размер страницы.
- **boolean nextPage()**
- **boolean previousPage()**

  Загружают следующую или предыдущую страницу строк. Возвращают логическое значение **true**, если существует следующая или предыдущая страница.
- **void acceptChanges()**
- **void acceptChanges(Connection *conn*)**

  Повторно подключаются к базе данных и записывают в нее изменения, внесенные в набор строк. Если с момента заполнения набора содержимое базы данных изменилось, данные не могут быть записаны в нее обратно. В этом случае генерируется исключение типа **SyncProviderException**.

javax.sql.rowset.RowSetProvider 7

- **static RowSetFactory newFactory()**
  Создает фабрику наборов строк.

javax.sql.rowset.RowSetFactory 7

- **CachedRowSet createCachedRowSet()**
- **FilteredRowSet createFilteredRowSet()**
- **JdbcRowSet createJdbcRowSet()**
- **JoinRowSet createJoinRowSet()**
- **WebRowSet createWebRowSet()**
  Создают набор строк заданного типа.

5.8. Метаданные

В предыдущих разделах рассматривались способы ввода и обновления содержимого таблиц базы данных и составления запросов. Помимо этого, в JDBC предусмотрены дополнительные возможности для получения сведений о *структуре* таблиц и самой базы данных. В частности, можно получить список всех таблиц базы данных или имена всех столбцов с типами данных в них. Эти сведения вряд ли будут полезны при разработке прикладной программы, предназначенной для работы с конкретной базой данных, потому что в таких случаях ее структура точно известна. Но они пригодятся тем разработчикам, которые создают свои программные продукты для работы с любыми базами данных.

В языке SQL сведения о структуре базы данных и ее компонентов называются *метаданными*. Такое название выбрано лишь для того, чтобы как-то отличать сведения о базе данных от ее основного содержимого. Существуют метаданные трех типов, описывающие структуру базы данных, структуру результирующих наборов и параметры подготовленных операторов.

Для получения более подробных сведений о структуре базы данных требуется объект типа `DatabaseMetaData`, который можно получить из установленного соединения с базой данных следующим образом:

```
DatabaseMetaData meta = conn.getMetaData();
```

Далее можно приступать непосредственно к получению метаданных. Так, если вызвать приведенный ниже метод, в итоге будет получен результирующий набор, содержащий сведения обо всех таблицах базы данных. (Параметры этого метода рассматриваются далее при описании соответствующего прикладного интерфейса API.)

```
ResultSet mrs = meta.getTables(null, null, null,
                    new String[] { "TABLE" });
```

Каждая строка из получаемого в итоге результирующего набора содержит сведения об отдельной таблице, а третий столбец в ней — имя таблицы, как поясняется далее в описании соответствующего прикладного интерфейса API. В приведенном ниже фрагменте кода организуется цикл для сбора сведений об именах всех таблиц в базе данных.

```
while (mrs.next())
  tableNames.addItem(mrs.getString(3));
```

Метаданные базы данных находят еще одно полезное применение. Базы данных могут иметь очень сложную структуру, а стандарт SQL предоставляет немало места для отклонений от нормы. Поэтому в интерфейсе `DatabaseMetaData` предусмотрено более сотни разных методов, которые можно использовать для получения сведений о структуре базы данных. Ниже приведены примеры вызова таких методов с довольно необычными именами. Судя по названиям этих методов, они предназначены главным образом для очень опытных разработчиков, в том числе и тех, кто занимается написанием переносимого кода, способного работать с разнотипными базами данных.

```
meta.supportsCatalogsInPrivilegeDefinitions()
```

и

```
meta.nullPlusNonNullIsNull()
```

Интерфейс `DatabaseMetaData` предоставляет сведения о самой базе данных, а сведения о результирующем наборе — интерфейс `ResultSetMetaData`. Получив результирующий набор по запросу, можно определить количество столбцов, имена столбцов, типы данных в них и ширину полей. Ниже приведен типичный цикл, в котором все эти сведения извлекаются с помощью соответствующих методов, выделенных полужирным.

```
ResultSet mrs = stat.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData meta = mrs.getMetaData();
for (int i = 1; i <= meta.getColumnCount(); i++)
{
  String columnName = meta.getColumnLabel(i);
  int columnWidth = meta.getColumnDisplaySize(i);
  . . .
}
```

В этом разделе поясняется, как создать простое инструментальное средство, предназначенное для просмотра и анализа структуры базы данных. Исходный код примера программы, реализующей это средство, приведен в листинге 5.4. В этой программе демонстрируется также применение кешированного набора строк.

В верхней части рабочего окна рассматриваемой здесь программы находится комбинированный список с именами всех таблиц базы данных. Как показано на рис. 5.6, после выбора какой-нибудь одной таблицы в центральной части фрейма будут представлены имена столбцов из этой таблицы, а также значения из первой строки. Для просмотра строк в таблице следует щелкнуть на кнопках Next (Следующая) и Previous (Предыдущая). Строки можно удалять, а также редактировать значения в них. Чтобы сохранить изменения в базе данных, следует щелкнуть на кнопке Save (Сохранить).

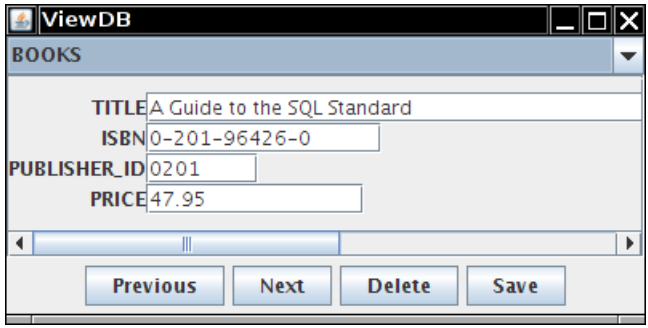


Рис. 5.6. Прикладная программа **ViewDB**

НА ЗАМЕТКУ! В состав баз данных обычно включаются инструментальные средства для просмотра и редактирования таблиц, обладающие намного большими возможностями. Если для вашей базы такое инструментальное средство отсутствует, попробуйте воспользоваться iSQL-Viewer (**http://isql.sourceforge.net**) или SQuirreL (**http://squirrel-sql.sourceforge.net**). Эти инструментальные средства позволяют просматривать таблицы любой базы данных, совместимой с прикладным интерфейсом JDBC. Рассматриваемая здесь программа отнюдь не претендует на то, чтобы соперничать с этими инструментальными средствами, и лишь демонстрирует общие принципы создания программ для работы с произвольными таблицами базы данных.

Листинг 5.4. Исходный код из файла **view/ViewDB.java**

```
1   package view;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.io.*;
6   import java.nio.file.*;
7   import java.sql.*;
8   import java.util.*;
9
10  import javax.sql.*;
11  import javax.sql.rowset.*;
12  import javax.swing.*;
13
14  /**
15   * В этой программе демонстрируется применение
16   * метаданных для отображения произвольно выбираемых
17   * таблиц в базе данных
18   * @version 1.34 2018-05-01
19   * @author Cay Horstmann
20   */
21  public class ViewDB
22 {
23    public static void main(String[] args)
24    {
25      EventQueue.invokeLater(() ->
26       {
```

```
27          var frame = new ViewDBFrame();
28          frame.setTitle("ViewDB");
29          frame.setDefaultCloseOperation(
30                JFrame.EXIT_ON_CLOSE);
31          frame.setVisible(true);
32        });
33    }
34  }
35
36  /**
37   * Фрейм, содержащий панель с кнопками для
38   * перемещения по данным
39   */
40  class ViewDBFrame extends JFrame
41  {
42    private JButton previousButton;
43    private JButton nextButton;
44    private JButton deleteButton;
45    private JButton saveButton;
46    private DataPanel dataPanel;
47    private Component scrollPane;
48    private JComboBox<String> tableNames;
49    private Properties props;
50    private CachedRowSet crs;
51    private Connection conn;
52
53    public ViewDBFrame()
54    {
55      tableNames = new JComboBox<String>();
56
57      try
58      {
59        readDatabaseProperties();
60        conn = getConnection();
61        DatabaseMetaData meta = conn.getMetaData();
62        try (ResultSet mrs = meta.getTables(null, null,
63             null, new String[] { "TABLE" }))
64        {
65          while (mrs.next())
66            tableNames.addItem(mrs.getString(3));
67        }
68      }
69      catch (SQLException ex)
70      {
71        for (Throwable t : ex)
72          t.printStackTrace();
73      }
74      catch (IOException ex)
75      {
76        ex.printStackTrace();
77      }
78
79      tableNames.addActionListener(
80        event -> showTable((String)
81                  tableNames.getSelectedItem(), conn));
82      add(tableNames, BorderLayout.NORTH);
```

```
83      addWindowListener(new WindowAdapter()
84        {
85          public void windowClosing(WindowEvent event)
86          {
87            try
88            {
89              if (conn != null) conn.close();
90            }
91            catch (SQLException ex)
92            {
93              for (Throwable t : ex)
94                t.printStackTrace();
95            }
96          }
97        });
98
99      var buttonPanel = new JPanel();
100     add(buttonPanel, BorderLayout.SOUTH);
101
102     previousButton = new JButton("Previous");
103     previousButton.addActionListener(
104                   event -> showPreviousRow());
105     buttonPanel.add(previousButton);
106
107     nextButton = new JButton("Next");
108     nextButton.addActionListener(
109                event -> showNextRow());
110     buttonPanel.add(nextButton);
111
112     deleteButton = new JButton("Delete");
113     deleteButton.addActionListener(
114                  event -> deleteRow());
115     buttonPanel.add(deleteButton);
116
117     saveButton = new JButton("Save");
118     saveButton.addActionListener(
119                event -> saveChanges());
120     buttonPanel.add(saveButton);
121     if (tableNames.getItemCount() > 0)
122     showTable(tableNames.getItemAt(0), conn);
123   }
124
125   /**
126    * Подготавливает текстовые поля для показа новой
127    * таблицы и отображает первую ее строку
128    * @param tableName Имя отображаемой таблицы
129    * @param conn Подключение к базе данных
130    */
131   public void showTable(String tableName,
132                         Connection conn)
133   {
134     try (Statement stat = conn.createStatement();
135          ResultSet result = stat.executeQuery(
136          "SELECT * FROM " + tableName))
137     {
138       // получить результирующий набор и
```

```
139       // скопировать его в кешируемый
140       // результирующий набор
141       RowSetFactory factory =
142                     RowSetProvider.newFactory();
143       crs = factory.createCachedRowSet();
144       crs.setTableName(tableName);
145       crs.populate(result);
146
147       if (scrollPane != null) remove(scrollPane);
148       dataPanel = new DataPanel(crs);
149       scrollPane = new JScrollPane(dataPanel);
150       add(scrollPane, BorderLayout.CENTER);
151       pack();
152       showNextRow();
153     }
154     catch (SQLException ex)
155     {
156       for (Throwable t : ex)
157         t.printStackTrace();
158     }
159   }
160
161   /**
162    * Осуществляет переход к предыдущей строке таблицы
163    */
164   public void showPreviousRow()
165   {
166     try
167     {
168       if (crs == null || crs.isFirst()) return;
169       crs.previous();
170       dataPanel.showRow(crs);
171     }
172     catch (SQLException ex)
173     {
174       for (Throwable t : ex)
175         t.printStackTrace();
176     }
177   }
178
179   /**
180    * Осуществляет переход к следующей строке таблицы
181    */
182   public void showNextRow()
183   {
184     try
185     {
186       if (crs == null || crs.isLast()) return;
187       crs.next();
188       dataPanel.showRow(crs);
189     }
190     catch (SQLException ex)
191     {
192       for (Throwable t : ex)
193          t.printStackTrace();
194     }
```

```
195   }
196
197   /**
198    * Удаляет строку из текущей таблицы
199    */
200   public void deleteRow()
201   {
202     if (crs == null) return;
203     new SwingWorker<Void, Void>()
204     {
205       public Void doInBackground() throws SQLException
206       {
207         crs.deleteRow();
208         crs.acceptChanges(conn);
209         if (crs.isAfterLast())
210           if (!crs.last()) crs = null;
211         return null;
212       }
213       public void done()
214       {
215         dataPanel.showRow(crs);
216       }
217     }.execute();
218   }
219   /**
220    * Сохраняет все внесенные изменения
221    */
222   public void saveChanges()
223   {
224     if (crs == null) return;
225     new SwingWorker<Void, Void>()
226     {
227       public Void doInBackground() throws SQLException
228       {
229         dataPanel.setRow(crs);
230         crs.acceptChanges(conn);
231         return null;
232       }
233     }.execute();
234   }
235
236   private void readDatabaseProperties()
237           throws IOException
238   {
239     props = new Properties();
240     try (InputStream in = Files.newInputStream(
241          Paths.get("database.properties")))
242     {
243       props.load(in);
244     }
245     String drivers = props.getProperty(
246                              "jdbc.drivers");
247     if (drivers != null)
248       System.setProperty("jdbc.drivers", drivers);
249   }
250
```

```
251    /**
252     * Получает сведения о подключении к базе данных из
253     * свойств, задаваемых в файле database.properties,
254     * и на их основании подключается к базе данных
255     * @return Подключение к базе данных
256     */
257    private Connection getConnection()
258          throws SQLException
259    {
260      String url = props.getProperty("jdbc.url");
261      String username = props.getProperty(
262                              "jdbc.username");
263      String password = props.getProperty(
264                              "jdbc.password");
265
266      return DriverManager.getConnection(url,
267                           username, password);
268    }
269 }
270
271 /**
272  * Панель для отображения содержимого
273  * результирующего набора
274  */
275 class DataPanel extends JPanel
276 {
277    private java.util.List<JTextField> fields;
276
277    /**
278     * Конструирует панель для отображения данных
279     * @param rs Результирующий набор, содержимое
280     *           которого отображается на данной панели
281     */
282    public DataPanel(RowSet rs) throws SQLException
283    {
284      fields = new ArrayList<>();
285      setLayout(new GridBagLayout());
286      var gbc = new GridBagConstraints();
287      gbc.gridwidth = 1;
288      gbc.gridheight = 1;
289
290      ResultSetMetaData rsmd = rs.getMetaData();
291      for (int i = 1; i <= rsmd.getColumnCount(); i++)
292      {
293        gbc.gridy = i - 1;
294
295        String columnName = rsmd.getColumnLabel(i);
296        gbc.gridx = 0;
297        gbc.anchor = GridBagConstraints.EAST;
298        add(new JLabel(columnName), gbc);
299
300        int columnWidth = rsmd.getColumnDisplaySize(i);
301        var tb = new JTextField(columnWidth);
302        if (!rsmd.getColumnClassName(i)
303                 .equals("java.lang.String"))
304          tb.setEditable(false);
```

```
305
306         fields.add(tb);
307
308         gbc.gridx = 1;
309         gbc.anchor = GridBagConstraints.WEST;
310         add(tb, gbc);
311       }
312     }
313
314     /**
315      * Отображает строку из таблицы базы данных,
316      * заполняя все текстовые поля значениями из столбцов
317      */
318     public void showRow(ResultSet rs)
319     {
320       try
321       {
322         if (rs == null) return;
323         for (int i = 1; i <= fields.size(); i++)
324         {
325           String field = rs ==
326                  null ? "" : rs.getString(i);
327           JTextField tb = fields.get(i - 1);
328           tb.setText(field);
329         }
330       }
331       catch (SQLException ex)
332       {
333         for (Throwable t : ex)
334           t.printStackTrace();
335       }
336     }
337
338     /**
339      * Обновляет измененными данными текущую
340      * строку из результирующего набора
341      */
342     public void setRow(RowSet rs) throws SQLException
343     {
344       for (int i = 1; i <= fields.size(); i++)
345       {
346         String field = rs.getString(i);
347         JTextField tb = fields.get(i - 1);
348         if (!field.equals(tb.getText()))
349           rs.updateString(i, tb.getText());
350       }
351       rs.updateRow();
352     }
353 }
```

java.sql.Connection 1.1

- **DatabaseMetaData getMetaData()**
  Возвращает метаданные в виде объекта типа **DatabaseMetaData** для подключения к базе данных.

java.sql.DatabaseMetaData 1.1

- **ResultSet getTables(String *catalog*, String schemaPattern, String *tableNamePattern*, String *types*[])**

  Возвращает из указанного каталога описание всех таблиц, совпадающих с шаблонами схемы и имен таблиц, а также с заданными критериями типов. (*Схема* описывает группу связанных вместе таблиц и полномочия доступа к ним, а *каталог* — группу связанных вместе схем. Эти понятия важны для структурирования крупных баз данных.)

  В качестве параметров ***catalog*** и **schemaPattern** могут быть указаны пустые символьные строки (""), чтобы извлечь таблицы без каталога и схемы, или же пустые значения **null**, если требуется возвратить таблицы независимо от каталога или схемы.

- Массив ***types*** содержит следующие имена типов таблиц: **TABLE**, **VIEW**, **SYSTEM TABLE**, **GLOBAL TEMPORARY**, **LOCAL TEMPORARY**, **ALIAS** и **SYNONYM**. Если вместо массива ***types*** указано пустое значение **null**, возвращаются таблицы всех типов.

  Результирующий набор состоит из пяти столбцов типа **String**, как показано ниже.

Столбец	Имя	Описание
1	**TABLE_CAT**	Каталог таблиц (может иметь пустое значение **null**)
2	**TABLE_SCHEM**	Схема (может иметь пустое значение **null**)
3	**TABLE_NAME**	Имя таблицы
4	**TABLE_TYPE**	Имя таблицы
5	**REMARKS**	Комментарии к таблице

- **int getJDBCMajorVersion() 1.4**
- **int getJDBCMinorVersion() 1.4**

  Возвращают основной и дополнительный номера версии драйвера JDBC, устанавливающего соединение с базой данных. Например, драйвер JDBC 3.0 имеет основной номер версии **3** и дополнительный номер версии **0**.

- **int getMaxConnections()**

  Возвращает максимальное количество подключений к базе данных, которые допускается производить одновременно.

- **int getMaxStatements()**

  Возвращает максимальное количество операторов SQL, которое допускается при каждом подключении к базе данных. Если это количество неограничено или неизвестно, то возвращается нулевое значение.

java.sql.ResultSet 1.1

- **ResultSetMetaData getMetaData()**

  Возвращает метаданные, связанные со столбцами текущего результирующего набора типа **ResultSet**.

java.sql.ResultSetMetaData 1.1

- **int getColumnCount()**
  Возвращает количество столбцов для текущего результирующего набора типа **ResultSet**.
- **int getColumnDisplaySize(int *column*)**
  Возвращает максимальную ширину столбца по указанному целочисленному индексу.
- **String getColumnLabel(int *column*)**
  Возвращает предполагаемый заголовок для указанного столбца.
- **String getColumnName(int *column*)**
  Возвращает имя столбца по указанному целочисленному индексу.

5.9. Транзакции

Группа команд может быть оформлена в виде *транзакции*, которая может быть *зафиксирована* после успешного выполнения всех операторов SQL или *откачена*, если при выполнении хотя бы одного из операторов произойдет какая-нибудь ошибка. Основной причиной для группирования операторов в транзакции служит сохранение *целостности базы данных*.

Допустим, требуется перевести денежные средства с одного банковского счета на другой. Для этого следует одновременно снять нужную сумму денег с одного счета и пополнить ею другой счет. Если после снятия суммы денег с одного счета, но перед пополнением другого произойдет системная ошибка, то операция с первым счетом должна быть отменена.

Операторы обновления базы данных могут быть сгруппированы в одну транзакцию. Если транзакция завершается полностью, то возможна ее *фиксация*. Если она не завершается полностью из-за каких-нибудь ошибок, то производится ее *откат*, т.е. отменяются все изменения в базе данных, которые выполнялись после предыдущей зафиксированной транзакции.

5.9.1. Программирование транзакций средствами JDBC

По умолчанию соединение с базой данных находится в режиме *автоматической фиксации*, т.е. результат выполнения каждого оператора SQL фиксируется в базе данных после успешного завершения этой команды. Как только оператор будет зафиксирован, откатить его уже нельзя. Для отключения режима автоматической фиксации можно вызвать следующий метод:

```
conn.setAutoCommit(false);
```

Отключив автоматическую фиксацию, можно приступать к созданию объекта типа `Statement` обычным образом:

```
Statement stat = conn.createStatement();
```

Затем метод `executeUpdate()` вызывается нужное количество раз, как показано ниже.

```
stat.executeUpdate(оператор1);
stat.executeUpdate(оператор2);
stat.executeUpdate(оператор3);
. . .
```

Если все эти операторы завершатся успешно, то результаты их выполнения фиксируются. Для этого вызывается метод `commit()` следующим образом:

```
conn.commit();
```

Если при выполнении любого из этих операторов произойдет ошибка, то производится откат всей транзакции. И для этого вызывается метод `rollback()` следующим образом:

```
conn.rollback();
```

При этом автоматически отменяются все операторы, выполнявшиеся после фиксации последней транзакции. Откат обычно производится в том случае, если при выполнении транзакции генерируется исключение типа `SQLException`.

5.9.2. Точки сохранения

Некоторые драйверы позволяют повысить уровень контроля над процессом отката с помощью *точек сохранения*. При создании точки сохранения отмечается точка, в которую можно впоследствии вернуться, не отменяя транзакцию в целом. В приведенном ниже фрагменте кода показано, как это осуществляется на практике.

```
// начать транзакцию; переход в данную точку происходит
// при вызове метода rollback():
Statement stat = conn.createStatement();
stat.executeUpdate(оператор1);
// установить точку сохранения; переход в эту точку
// происходит при вызове метода rollback(svpt):
Savepoint svpt = conn.setSavepoint();

stat.executeUpdate(оператор2);
if (. . .)
  // отменить результат выполнения оператора2:
  conn.rollback(svpt);
. . .
conn.commit();
```

Если точка сохранения больше не нужна, ее следует освободить:

```
conn.releaseSavepoint(svpt);
```

5.9.3. Групповые обновления

Допустим, в прикладной программе требуется выполнить много операторов `INSERT` для заполнения таблицы базы данных. Повысить производительность такой программы можно с помощью *группового обновления*. При групповом обновлении операторы SQL собираются вместе и передаются на выполнение группой, а не по отдельности.

НА ЗАМЕТКУ! Чтобы выяснить, поддерживается ли в базе данных групповое обновление, достаточно вызвать метод **`supportsBatchUpdates()`** из интерфейса **`DatabaseMetaData`**.

Помимо операторов управления данными INSERT, UPDATE и DELETE, для группового обновления можно также использовать операторы определения данных, в том числе CREATE TABLE и DROP TABLE. Но для этой цели не подходит оператор SELECT, поскольку его выполнение в группе с другими операторами приводит к исключению. (В принципе оператор SELECT не имеет смысла выполнять в групповом режиме, поскольку он возвращает результирующий набор, не обновляя базу данных.)

Для группового обновления сначала создается объект типа Statement:

```
Statement stat = conn.createStatement();
```

Затем вместо метода executeUpdate() вызывается метод addBatch():

```
String command = "CREATE TABLE . . ."
stat.addBatch(оператор);

while (. . .)
{
  command = "INSERT INTO . . . VALUES (" + . . . + ")";
  stat.addBatch(оператор);
}
```

Наконец, все операторы SQL передаются вместе для группового обновления базы данных, как показано ниже. Метод executeBatch() возвращает массив подсчетов строк, обработанных при выполнении каждого оператора из данной группы.

```
int[] counts = stat.executeBatch();
```

Для правильной обработки ошибок в групповом режиме групповое обновление следует рассматривать как единую транзакцию. Если в ходе группового обновления произойдет сбой или возникнет ошибка, следует произвести откат в исходное состояние.

Прежде всего следует отключить режим автоматической фиксации, собрать операторы SQL в группу, выполнить их и зафиксировать результаты, а затем восстановить режим автоматической фиксации, как выделено полужирным в приведенном ниже фрагменте кода.

```
boolean autoCommit = conn.getAutoCommit();
conn.setAutoCommit(false);
Statement stat = conn.getStatement();
. . .
// продолжать вызовы метода stat.addBatch(. . .);
. . .
stat.executeBatch();
conn.commit();
conn.setAutoCommit(autoCommit);
```

java.sql.Connection 1.1

- **boolean getAutoCommit()**
- **void setAutoCommit(boolean *b*)**

  Получают или устанавливают режим автоматической фиксации для данного подключения к базе данных. Если параметр ***b*** принимает логическое значение **true**, результаты выполнения всех операторов SQL автоматически фиксируются после их завершения.

java.sql.Connection 1.1 *(окончание)*

- **void commit()**
  Фиксирует все операторы SQL, которые были выполнены с момента последней фиксации.
- **void rollback()**
  Производит откат, отменяя все изменения, которые были внесены с момента последней фиксации.
- **Savepoint setSavepoint()** 1.4
- **Savepoint setSavepoint(String *name*)** 1.4
  Устанавливают безымянную или именованную точку сохранения.
- **void rollback(Savepoint *svpt*)** 1.4
  Производит откат всех операторов SQL до указанной точки сохранения.
- **void releaseSavepoint(Savepoint *svpt*)** 1.4
  Освобождает указанную точку сохранения.

java.sql.Savepoint 1.4

- **int getSavepointId()**
  Возвращает идентификатор данной безымянной точки сохранения. Если данная точка оказывается именованной, генерируется исключение типа **SQLException**.
- **String getSavepointName()**
  Возвращает имя точки сохранения. Если данная точка оказывается безымянной, генерируется исключение типа **SQLException**.

java.sql.Statement 1.1

- **void addBatch(String *command*)** 1.2
  Включает указанный оператор SQL как выполняемую команду в текущую группу для группового обновления.
- **int[] executeBatch()** 1.2
- **long[] executeLargeBatch()** 8
  Выполняют все операторы SQL из текущей группы. Каждое значение в возвращаемом массиве соответствует одному из операторов, входящих в данную группу. Если это положительное значение, то оно обозначает подсчет обновленных строк. Если же это значение **SUCCESS_NO_INFO**, то оно обозначает, что оператор удалось выполнить, но подсчет обновленных строк недоступен. Если это значение **EXECUTE_FAILED**, то оно обозначает, что выполнить оператор не удалось.

java.sql.DatabaseMetaData 1.1

- **boolean supportsBatchUpdates()** 1.2
  Возвращает логическое значение **true**, если драйвер поддерживает групповое обновление.

5.9.4. Расширенные типы данных SQL

В табл. 5.8 перечислены все типы данных SQL, поддерживаемых в JDBC, а также их эквиваленты в Java.

Таблица 5.8. Типы данных SQL и соответствующие им типы в Java

Тип данных SQL	Тип данных Java
`INTEGER` или `INT`	`int`
`SMALLINT`	`short`
`NUMERIC(m, n)`, `DECIMAL(m, n)` или `DEC(m, n)`	`java.math.BigDecimal`
`FLOAT(n)`	`double`
`REAL`	`float`
`DOUBLE`	`double`
`CHARACTER(n)` или `CHAR(n)`	`String`
`VARCHAR(n)`, `LONG VARCHAR`	`String`
`BOOLEAN`	`boolean`
`DATE`	`java.sql.Date`
`TIME`	`java.sql.Time`
`TIMESTAMP`	`java.sql.Timestamp`
`BLOB`	`java.sql.Blob`
`CLOB`	`java.sql.Clob`
`ARRAY`	`java.sql.Array`
`ROWID`	`java.sql.RowId`
`NCHAR(n)`, `NVARCHAR(n)`, `LONG NVARCHAR`	`String`
`NCLOB`	`java.sql.NClob`
`SQLXML`	`java.sql.SQLXML`

Тип `ARRAY` представляет в SQL последовательность значений. Например, таблица `Student` может иметь столбец с оценками `Scores` типа `ARRAY OF INTEGER`, т.е. с массивом целочисленных значений. Метод `getArray()` возвращает объект интерфейса типа `java.sql.Array`. В этом интерфейсе предусмотрены также методы извлечения значений из массива.

При получении большого объекта (LOB) или массива из базы данных конкретное содержимое извлекается из нее только после запроса отдельных значений. Это сделано для повышения эффективности работы с базой данных, поскольку большой объект может оказаться довольно объемистым.

В некоторых базах данных поддерживаются значения типа `ROWID`, описывающие местонахождение строки, что позволяет очень быстро извлечь ее из таблицы. В версии JDBC 4 внедрен интерфейс `java.sql.RowId`, предоставляющий методы для ввода идентификатора строки в запросы и извлечения его из получаемых результатов.

Строка национальных символов (типа NCHAR и его вариантов) служит для хранения символьных строк в локальной кодировке, а также для их сортировки по заданным условиям локальной сортировки. В версии JDBC 4 предоставляются методы для взаимного преобразования объектов Java типа String и строк национальных символов в запросах и получаемых результатах.

В некоторых базах данных допускается хранение данных, типы которых определяются пользователем. В версии JDBC 3 поддерживается механизм автоматического преобразования структурированных типов данных SQL в объекты Java. Кроме того, в некоторых базах данных предоставляется собственный механизм хранения данных в формате XML. В версии JDBC 4 внедрен интерфейс SQLXML, который может служить связующим звеном между внутренним представлением данных в формате XML и интерфейсами Source/Result модели DOM, а также двоичными потоками ввода-вывода. Дополнительные сведения об интерфейсе SQLXML можно найти в документации на соответствующий прикладной интерфейс API.

На этом рассмотрение расширенных типов данных SQL в этой главе завершается. Дополнительные сведения по этому вопросу можно найти в упоминавшейся ранее книге *JDBC™ API Tutorial and Reference, Third Edition* и спецификации интерфейса JDBC 4.

5.10. Управление подключением к базам данных в веб-приложениях и корпоративных приложениях

Описанный ранее способ соединения с базой данных с помощью параметров из файла свойств database.properties подходит только для очень простых тестовых программ и совершенно не годится для крупномасштабных приложений. При установке приложения JDBC в корпоративной среде соединения с базами данных поддерживаются через JNDI (Java Naming and Directory Interface — интерфейс для служб каталогов и именования в Java). Свойства источников данных в пределах всего предприятия хранятся в отдельном каталоге. Благодаря этому обеспечивается централизованное управление именами пользователей, паролями и URL в JDBC. В такой среде для подключения к базе данных рекомендуется использовать код, подобный следующему:

```
var jndiContext = new InitialContext();
var source = (DataSource) jndiContext.lookup(
             "java:comp/env/jdbc/corejava");
Connection conn = source.getConnection();
```

Обратите внимание на то, что в этом коде уже не используется диспетчер драйверов типа DriverManager. Вместо него для поиска *источника данных* применяется служба JNDI. В качестве источника данных служит интерфейс DataSource, позволяющий устанавливать простые соединения типа JDBC, а также выполнять ряд более сложных функций, например, распределенные транзакции с несколькими базами данных. Интерфейс DataSource входит в пакет javax.sql, расширяющий стандартную библиотеку Java.

НА ЗАМЕТКУ! В контейнере Java EE не нужно даже программировать поиск в службе JNDI. Достаточно сделать следующую аннотацию **`@Resource`** к полю типа **`DataSource`**, чтобы во время загрузки приложения был автоматически задан эталонный источник данных:

```
@Resource(name="jdbc/corejava")
private DataSource source;
```

Очевидно, что источник данных нуждается в настройке. Так, если прикладная программа для работы с базой данных должна выполняться в контейнере сервлетов (например, Apache Tomcat) или на сервере приложений (например, GlassFish), то сведения о настройке базы данных (в том числе имя службы JNDI, URL в JDBC, имя пользователя и пароль) целесообразно разместить в конфигурационном файле или задать в графическом пользовательском интерфейсе администратора.

Управление именами пользователей и регистрационными данными — это лишь один из вопросов, требующих особого внимания. Второй вопрос связан со стоимостью подключения к базам данных. В примерах программ, представленных в этой главе, применяются две методики для получения требующегося соединения с базой данных. Так, в самом начале программы `QueryDB` из листинга 5.3 устанавливается единственное соединение с базой данных, которое разрывается по завершении программы. В программе `ViewDB` из листинга 5.4 новое соединение устанавливается всякий раз, когда в нем возникает потребность.

Но ни одну из этих методик нельзя считать удовлетворительной. Ведь соединения с базами данных — это конечный ресурс. И если пользователь приостановит работу с приложением на некоторое время, то соединение не следует оставлять установленным. С другой стороны, получение соединения для каждого запроса и последующий его разрыв обходится очень дорого.

В качестве выхода из этого положения целесообразно организовать *пул* соединений. Это означает, что соединения с базами данных не разрываются физически, а сохраняются в очереди и повторно используются для других запросов. Организация пулов соединений является важной служебной функцией, и поэтому в спецификации JDBC предоставляются вспомогательные средства для ее реализации. Однако в различных базах данных пул соединений может быть реализован по-разному. Причем он не всегда входит в состав драйвера JDBC. Некоторые поставщики веб-контейнеров и серверов приложений предлагают реализации пулов соединений в составе своих приложений.

Использование пула соединений полностью прозрачно для программиста. Чтобы извлечь соединение из пула, достаточно получить соответствующий источник данных и вызвать метод `getConnection()`. По окончании работы с базой данных через это соединение следует вызвать метод `close()`. При этом соединение физически не разрывается, но в пул соединений поступает сообщение, что соединение больше не требуется. Как правило, в пуле соединений принимаются необходимые меры, чтобы сохранить в нем и подготовленные операторы.

Итак, вы ознакомились с самыми основами JDBC, которые требуются для разработки простых прикладных программ, взаимодействующих с базами данных. Но, как отмечалось в начале этой главы, базы данных могут иметь очень сложную структуру, а более сложные вопросы работы с ними выходят за рамки рассмотрения данной книги. Поэтому интересующихся подобными вопросами еще раз отсылаем к упоминавшейся ранее книге *JDBC™ API Tutorial and Reference, Third Edition* и спецификации JDBC.

Из этой главы вы узнали о том, как организуется взаимодействие с реляционными базами данных в Java. Следующая глава посвящена библиотеке даты и времени, внедренной в версии Java 8.

ГЛАВА 6

Прикладной интерфейс API даты и времени

В этой главе...

- Временная шкала
- Местные даты
- Корректоры дат
- Местное время
- Поясное время
- Форматирование и синтаксический анализ даты и времени
- Взаимодействие с унаследованным кодом

Время летит как стрела, и мы можем легко установить начальный момент, чтобы отсчитывать время вперед и назад. Так почему же так трудно обращаться со временем? Все дело в самих людях. Не проще ли было, если бы мы обращались друг к другу следующим образом: "Встретимся в 1371409200, только не опаздывай!" Но ведь нам нужно соотносить время с конкретным временем суток и года. Именно здесь и возникают трудности. В версии Java 1.0 имелся класс `Date`, реализация которого теперь кажется наивной, а большинство его методов стали нерекомендованными к употреблению в версии Java 1.1, в которой был внедрен класс `Calendar`. Безусловно, его прикладной интерфейс API не был совершенным, его экземпляры были изменяемыми, и в нем не учитывались потерянные секунды. Более совершенным оказался прикладной интерфейс API даты и времени из пакета `java.time`, внедренный в версии Java 8. В нем были устранены недостатки прошлых реализаций, и можно надеяться, что он послужит нам еще немало времени. В этой главе будет показано, что именно делает расчеты времени столь

неприятными и как подобные трудности разрешаются в прикладном интерфейсе API даты и времени.

6.1. Временная шкала

По традиции основополагающей единицей отсчета времени является секунда, производная от вращения Земли вокруг своей оси. Полный оборот Земля совершает за 24 часа, или 24 × 60 × 60 = 86400 секунд, и поэтому точное определение секунды кажется делом астрономических измерений. К сожалению, Земля испытывает незначительные колебания при вращении, что потребовало более точного определения секунды. И такое определение было сформулировано в 1967 году. Оно вполне согласуется с исторически сложившимся определением и в то же время основывается на внутреннем свойстве атомов Цезия-133. С тех пор официальное время хранится в разветвленной сети атомных часов.

Нередко официальные часы-хранители времени синхронизируют абсолютное время с вращением Земли. Прежде официальные секунды подвергались незначительной коррекции, но с 1972 года стали периодически вводиться так называемые "потерянные" секунды. (Теоретически секунду можно было бы время от времени удалять, но этого так и не произошло.) В настоящее время снова ведутся дискуссии об изменении системы отсчета времени. Очевидно, что причиной тому служат потерянные секунды, и поэтому во многих вычислительных системах применяется так называемое "сглаживание" там, где время искусственно замедляется или ускоряется перед потерянной секундой, чтобы сохранить ровно 86400 секунд в сутках. Такой способ вполне работоспособен, поскольку местное время на компьютере отсчитывается не совсем точно, а компьютеры обычно синхронизируются с внешней службой времени.

В соответствии со спецификацией на прикладной интерфейс API даты и времени в Java требуется временная шкала, которая

- имеет 86400 секунд в сутках;
- точно соответствует официальному времени в полдень каждого дня;
- близко соответствует ему в другое время суток строго определенным способом.

Благодаря этому язык Java может гибко подстраиваться к будущим изменениям в отсчете официального времени. В языке Java класс `Instant` представляет точку на временной шкале. Исходная точка отчета времени, называемая *эпохой*, произвольно задана в полночь 1 января 1970 года на нулевом меридиане, проходящем через Гринвичскую королевскую обсерваторию в Лондоне. Аналогичное соглашение принято и для отсчета времени по стандарту POSIX в системе Unix. Начиная с исходной точки отчета, время измеряется в секундах, как вперед, так и назад, с точностью до наносекунд, а каждые сутки составляют 86400 секунд. При отсчете времени назад значения типа `Instant` достигают миллиарда лет (т.е. минимальной точки `Instant.MIN` на временной шкале). И хотя этого недостаточно, чтобы выразить возраст вселенной (около 13,5 млрд лет), тем не менее, должно быть достаточно для практических целей. Ведь миллиард лет назад Земля была

покрыта льдом и населялась микроскопическими предшественниками современных растений и животных. А максимальная точка на временной шкале (`Instant.MAX`) соответствует 31 декабря 1000000000 года.

В результате вызова статического метода `Instant.now()` получается текущий момент времени. Два момента времени можно сравнить с помощью методов `equals()` и `compareTo()` обычным образом, чтобы использовать моменты времени как его отметки.

Для определения разности двух моментов времени служит статический метод `Duration.between()`. В качестве примера ниже показано, как измерить текущее время выполнения алгоритма.

```
Instant start = Instant.now();
runAlgorithm();
Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);
long millis = timeElapsed.toMillis();
```

Объект типа `Duration` определяет промежуток между двумя моментами времени. Длительность промежутка типа `Duration` в обычных единицах измерения времени можно получить, вызвав метод `toNanos()`, `toMillis()`, `toSeconds()`, `toMinutes()`, `toHours()` или `toDays()`.

Стремясь рассчитать время с точностью до наносекунд, не следует забывать о возможном переполнении. Так, значение типа `long` может содержат количество наносекунд порядка 300 лет. Если короткие промежутки времени можно просто преобразовать в наносекунды, то для длительных промежутков времени лучше воспользоваться объектом типа `Duration`, храня количество секунд в поле типа `long`, а количество наносекунд — в поле типа `int`. Для выполнения арифметических операций над промежутками времени в классе `Duration` предоставляется целый ряд методов, представленных в конце этого раздела.

Так, если требуется проверить, выполняется ли один алгоритм, по крайней мере в десять раз быстрее, чем другой, достаточно выполнить следующие расчеты:

```
Duration timeElapsed2 = Duration.between(start2, end2);
boolean overTenTimesFaster = timeElapsed.multipliedBy(10)
                  .minus(timeElapsed2).isNegative();
```

Данный пример демонстрирует лишь синтаксис для расчетов времени. Но поскольку алгоритмы не могут длиться сотни лет, приведенные выше расчеты времени их выполнения можно упростить следующим образом:

```
boolean overTenTimesFaster =
    timeElapsed.toNanos() * 10 < timeElapsed2.toNanos();
```

НА ЗАМЕТКУ! Классы **`Instant`** и **`Duration`** неизменяемы, поэтому все методы вроде **`multipliedBy()`** или **`minus()`** возвращают новые экземпляры этих классов.

В примере программы из листинга 6.1 демонстрируется применение классов `Instant` и `Duration` для расчета времени выполнения двух алгоритмов.

Листинг 6.1. Исходный код из файла **timeline/TimeLine.java**

```
1  package timeline;
2
3  /**
4   * @version 1.0 2016-05-10
5   * @author Cay Horstmann
6   */
7  import java.time.*;
8  import java.util.*;
9  import java.util.stream.*;
10
11 public class Timeline
12 {
13   public static void main(String[] args)
14   {
15     Instant start = Instant.now();
16     runAlgorithm();
17     Instant end = Instant.now();
18     Duration timeElapsed = Duration.between(start, end);
19     long millis = timeElapsed.toMillis();
20     System.out.printf("%d milliseconds\n", millis);
21
22     Instant start2 = Instant.now();
23     runAlgorithm2();
24     Instant end2 = Instant.now();
25     Duration timeElapsed2 =
26              Duration.between(start2, end2);
27     System.out.printf("%d milliseconds\n",
28                       timeElapsed2.toMillis());
29     boolean overTenTimesFaster =
30             timeElapsed.multipliedBy(10)
31            .minus(timeElapsed2).isNegative();
32     System.out.printf("The first algorithm is "
33                     + "%smore than ten times faster",
34                       overTenTimesFaster ? "" : "not ");
35   }
36
37   public static void runAlgorithm()
38   {
39     int size = 10;
40     List<Integer> list = new Random().ints()
41                   .map(i -> i % 100).limit(size)
42                   .boxed().collect(Collectors.toList());
43     Collections.sort(list);
44     System.out.println(list);
45   }
46
47   public static void runAlgorithm2()
48   {
49     int size = 10;
50     List<Integer> list = new Random().ints()
51                   .map(i -> i % 100).limit(size)
52                   .boxed().collect(Collectors.toList());
53     while (!IntStream.range(1, list.size())
54                      .allMatch(i -> list.get(i - 1)
55                      .compareTo(list.get(i)) <= 0))
```

```
56        Collections.shuffle(list);
57      System.out.println(list);
58    }
59 }
```

java.time.Instant 8

- **static Instant now()**

  Получает текущий момент времени из наилучших среди имеющихся системных часов.

- **Instant plus(TemporalAmount *amountToAdd*)**
- **Instant minus(TemporalAmount *amountToSubtract*)**

  Возвращают момент времени, отстоящий на указанную величину от данного момента времени, представленного объектом типа **Instant**. Интерфейс **TemporalAmount** реализуется в классах **Duration** и **Period** (см. далее раздел 6.2).

- **Instant (plus|minus)(Nanos|Millis|Seconds)(long *number*)**

  Возвращает момент времени, отстоящий на указанное количество наносекунд, миллисекунд или секунд от данного момента времени.

java.time.Duration 8

- **static Duration of(Nanos|Millis|Seconds|Minutes|Hours|Days)(long *number*)**

  Возвращает промежуток времени в количестве указанных единиц измерения.

- **static Duration between(Temporal *startInclusive*, Temporal *endExclusive*)**

  Возвращает промежуток времени между его указанными моментами. Интерфейс **Temporal** реализуется в классе **Instant**, а также в классах **LocalDate/LocalDateTime/LocalTime** (см. далее раздел 6.4) и **ZonedDateTime** (см. далее раздел 6.5).

- **long toNanos()**
- **long toMillis()**
- **long toSeconds()** 9
- **long toMinutes()**
- **long toHours()**
- **long toSeconds()**
- **long toSeconds()**
- **long toDays()**

  Получают промежуток времени, представленный объектом типа **Duration**, в количестве единиц измерения, обозначаемых в имени соответствующего метода.

- **int to(Nanos|Millis|Seconds|Minutes|Hours)Part()** 9
- **long to(Days|Hours|Minutes|Seconds|Millis|Nanos)Part()** 9

  Возвращают часть промежутка времени, представленного объектом типа **Duration**, в указанных единицах измерения. Например, промежуток времени 100 секунд состоит из 1 минуты и 40 секунд.

java.time.Duration 8 *(окончание)*

- **Instant plus(TemporalAmount *amountToAdd*)**
- **Instant minus(TemporalAmount *amountToSubtract*)**

  Возвращают момент времени, отстоящий от данного промежутка времени, представленного объектом типа **Instant**, на указанную величину. Интерфейс **TemporalAmount** реализуется в классах **Duration** и **Period** (см. далее раздел 6.2).
- **Duration multipliedBy(long *multiplicand*)**
- **Duration dividedBy(long *divisor*)**
- **Duration negated()**

  Возвращают промежуток времени, получаемый умножением или делением данного промежутка времени, представленного объектом типа **Duration**, на указанную величину или на **-1**.
- **boolean isZero()**
- **boolean isNegative()**

  Возвращают логическое значение **true**, если данный промежуток времени, представленный объектом типа **Duration**, оказывается нулевым или отрицательным.
- **Duration (plus|minus)(Nanos|Millis|Seconds|Minutes|Hours|Days)(long *number*)**

  Возвращает промежуток времени, получаемый в виде объекта типа **Duration** сложением или вычитанием заданного количества в указанных единицах измерения времени.

6.2. Местные даты

Теперь перейдем от абсолютного к обычному в обиходе времени. Такое время в прикладном интерфейсе Java API представлено двумя категориями *местного времени и даты* и *поясного времени*. Местное время и дата обозначают время суток или дату, но не связаны с часовым поясом. Примером местной даты служит 14 июня 1903 года (в этот день родился Алонсо Черч, изобретатель лямбда-вычислений). Эта дата не содержит ни время суток, ни часовой пояс и поэтому не соответствует точному моменту времени. С другой стороны, дата 16 июля 1969 года, 09:32:00 по восточному поясному времени (момент запуска космического корабля "Аполлон-11"), представляет точный момент времени на временной шкале с учетом часового пояса.

Во многих расчетах времени учитывать часовые пояса не требуется, а иногда они могут даже мешать. Допустим, требуется запланировать еженедельные совещания в 10:00. Если добавить 7 дней (т.е. 7 × 24 × 60 × 60 секунд) к последнему часовому поясу, то невольно можно пересечь границу, обозначающую переход на летнее или зимнее время, и тогда совещание состоится на час раньше или позже!

Именно по этой причине разработчики прикладного интерфейса API даты и времени рекомендуют не пользоваться поясным временем, кроме тех случаев, когда требуется представить экземпляры абсолютного времени. Дни рождения, праздники, сроки исполнения и прочие моменты времени лучше всего представить в виде местного времени и даты.

Объект типа LocalDate определяет местную дату с указанием года, месяца и дня месяца. Для построения этого объекта можно воспользоваться статическим методом now() или of(), как показано ниже.

```
LocalDate today = LocalDate.now(); // Текущая дата
LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);
alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);
    // Применяется перечисление Month
```

В отличие от нестандартных соглашений, принятых в системе Unix и классе java.util.Date, где отсчет месяцев начинается с нуля, а отсчет лет — с 1900 года, месяц года можно обозначить обычными числами. С другой стороны, для этой цели можно воспользоваться перечислением Month. Наиболее употребительные методы для манипулирования объектами типа LocalDate, представляющими местное время, представлены в конце этого раздела.

Например, День программиста приходится на 256-й день года. Ниже показано, насколько просто рассчитывается этот день.

```
// 13 сентября, но в високосный год — 12 сентября:
LocalDate programmersDay = LocalDate.of(2014, 1, 1)
                                    .plusDays(255);
```

Напомним, что разность двух моментов времени составляет промежуток, представленный объектом типа Duration. Для местных дат ему соответствует период времени, который представлен объектом типа Period и выражает количество прошедших лет, месяцев или дней. И хотя для получения местной даты дня своего рождения в следующем году достаточно сделать вызов birthday.plus(Period.ofYears(1)), в високосный год вряд ли получится правильный результат в результате вызова birthday.plus(Duration.ofDays(365)).

Метод until() возвращает разность двух местных дат. Например, в результате следующего вызова:

```
independenceDay.until(christmas)
```

получается период времени 5 месяцев и 21 день, что не очень удобно, поскольку количество дней в месяце варьируется. Поэтому для выявления количества дней лучше сделать следующий вызов:

```
independenceDay.until(christmas, ChronoUnit.DAYS) // 174 дня
```

ВНИМАНИЕ! Вызов некоторых методов из класса **LocalDate** мог бы привести к получению несуществующих дат. Так, если добавить один месяц к дате 31 января, то в конечном итоге не должна получиться дата 31 февраля. Вместо генерирования исключения эти методы возвращают последний достоверный день месяца. Например, в результате одного из следующих вызовов:

```
LocalDate.of(2016, 1, 31).plusMonths(1)
```

или

```
LocalDate.of(2016, 3, 31).minusMonths(1)
```

получается дата 29 февраля 2016.

Метод getDayOfWeek() возвращает день недели в виде соответствующего значения из перечисления DayOfWeek. В частности, понедельнику соответствует

значение `DayOfWeek.MONDAY`, равное `1`, а воскресенью — значение `DayOfWeek.SUNDAY`, равное `7`. Например, в результате следующего вызова:

```
LocalDate.of(1900, 1, 1).getDayOfWeek().getValue()
```

возвращается значение `1`. У перечисления `DayOfWeek` имеются удобные методы `plus()` и `minus()` для расчета дней недели по модулю 7. Так, в результате вызова `DayOfWeek.SATURDAY.plus(3)` возвращается значение `DayOfWeek.TUESDAY`.

НА ЗАМЕТКУ! Выходные дни фактически приходятся на конец недели. В то же время в классе **`java.util.Calendar`** воскресенью соответствует значение **1**, а субботе — значение **7**.

В версии Java 9 внедрены два удобных метода `datesUntil()`, возвращающих потоки объектов типа `LocalDate`:

```
LocalDate start = LocalDate.of(2000, 1, 1);
LocalDate endExclusive = LocalDate.now();
Stream<LocalDate> allDays = start.datesUntil(endExclusive);
Stream<LocalDate> firstDaysInMonth =
       start.datesUntil(endExclusive, Period.ofMonths(1));
```

Помимо класса `LocalDate`, имеются классы `MonthDay`, `YearMonth` и `Year` для описания частичных дат. Например, дата 25 декабря (с указанным годом) может быть представлена объектом класса `MonthDay`. Применение класса `LocalDate` демонстрируется в примере программы из листинга 6.2.

Listing 6.2. Исходный код из файла **`localdates/LocalDates.java`**

```
1  package localdates;
2
3  /**
4   * @version 1.0 2016-05-10
5   * @author Cay Horstmann
6   */
7  import java.time.*;
8  import java.time.temporal.*;
9  import java.util.stream.*;
10
11 public class LocalDates
12 {
13   public static void main(String[] args)
14   {
15     LocalDate today = LocalDate.now(); // Текущая дата
16     System.out.println("today: " + today);
17
18     LocalDate alonzosBirthday =
19               LocalDate.of(1903, 6, 14);
20     alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);
21     // Применяется перечисление Month
22     System.out.println("alonzosBirthday: "
23                        + alonzosBirthday);
24
25     LocalDate programmersDay = LocalDate.of(2018, 1, 1)
26                                         .plusDays(255);
```

```
27      // 13 сентября, но в високосный год 12 сентября
28      System.out.println("programmersDay: "
29                         + programmersDay);
30
31      LocalDate independenceDay =
32                LocalDate.of(2018, Month.JULY, 4);
33      LocalDate christmas =
34                LocalDate.of(2018, Month.DECEMBER, 25);
35
36      System.out.println("Until christmas: "
37                         + independenceDay.until(christmas));
38      System.out.println("Until christmas: "
39                         + independenceDay.until(
40                                christmas, ChronoUnit.DAYS));
41
42      System.out.println(LocalDate.of(2016, 1, 31)
43                .plusMonths(1));
44      System.out.println(LocalDate.of(2016, 3, 31)
45                .minusMonths(1));
46
47      DayOfWeek startOfLastMillennium =
48                LocalDate.of(1900, 1, 1).getDayOfWeek();
49      System.out.println("startOfLastMillennium: "
50                         + startOfLastMillennium);
51      System.out.println(startOfLastMillennium.getValue());
52      System.out.println(DayOfWeek.SATURDAY.plus(3));
53
54      LocalDate start = LocalDate.of(2000, 1, 1);
55      LocalDate endExclusive = LocalDate.now();
56      Stream<LocalDate> firstDaysInMonth =
57             start.datesUntil(endExclusive,
58                              Period.ofMonths(1));
59      System.out.println("firstDaysInMonth: "
60         + firstDaysInMonth.collect(Collectors.toList()));
61    }
62  }
```

java.time.LocalDate 8

- **static LocalDate now()**

  Получает объект типа **LocalDate**, представляющий текущую местную дату.
- **static LocalDate of(int *year*, int *month*, int *dayOfMonth*)**
- **static LocalDate of(int *year*, Month *month*, int *dayOfMonth*)**

  Возвращают местную дату по указанному году, месяцу (в виде целого числа в пределах от **1** до **12** или значения из перечисления **Month**) и дню месяца (в виде целого числа в пределах от **1** до **31**).
- **LocalDate (plus|minus)(Days|Weeks|Months|Years)(long *number*)**

  Возвращает местную дату в виде объекта типа **LocalDate**, получаемую сложением или вычитанием заданного количества в указанных единицах измерения времени.

java.time.LocalDate 8 *(окончание)*

- **LocalDate plus(TemporalAmount *amountToAdd*)**
- **LocalDate minus(TemporalAmount *amountToSubtract*)**

  Возвращают момент времени, отстоящий на указанную величину от данного момента времени. Интерфейс **TemporalAmount** реализуется в классах **Duration** и **Period**.
- **LocalDate withDayOfMonth(int *dayOfMonth*)**
- **LocalDate withDayOfYear(int *dayOfYear*)**
- **LocalDate withMonth(int *month*)**
- **LocalDate withYear(int *year*)**

  Возвращают новую местную дату в виде объекта типа **LocalDate** по указанному дню месяца, дню года, месяцу или году.
- **int getDayOfMonth()**

  Возвращает день месяца (в виде целого числа в пределах от **1** до **31**).
- **int getDayOfYear()**

  Возвращает день года (в виде целого числа в пределах от **1** до **366**).
- **DayOfWeek getDayOfWeek()**

  Возвращает день недели в виде значения из перечисления **DayOfWeek**.
- **Month getMonth()**
- **int getMonthValue()**

  Возвращают день месяца в виде значения из перечисления **Month** или целого числа в пределах от **1** до **12**.
- **int getYear()**

  Возвращает год в виде целого числа в пределах от **–999999999** до **999999999**.
- **Period until(ChronoLocalDate *endDateExclusive*)**

  Возвращает период вплоть до указанной конечной даты. Интерфейс **ChronoLocalDate** реализуется в классе **LocalDate**, а также в классах дат для календарей, отличающихся от григорианского.
- **boolean isBefore(ChronoLocalDate *other*)**
- **boolean isAfter(ChronoLocalDate *other*)**

  Возвращают логическое значение **true**, если текущая дата предшествует указанной дате или следует после нее.
- **boolean isLeapYear()**

  Возвращает логическое значение **true**, если год оказывается високосным, т.е. если он делится на **4**, но не на **100** или **400**. Этот алгоритм применяется ко всем предыдущим годам, хотя он исторически неточный. (Високосные годы были введены в 46 году до н.э., а правила деления на **100** или **400** — при реформе григорианского календаря в 1582 году. На повсеместное распространение этой реформы понадобилось более 300 лет.)
- **Stream<LocalDate> datesUntil(LocalDate *endExclusive*) 9**
- **Stream<LocalDate> datesUntil(LocalDate *endExclusive*, Period *step*) 9**

  Возвращают поток дат от текущей местной даты вплоть до указанной конечной даты с шагом **1** или заданным периодом.

java.time.Period 8

- **static Period of(int *years*, int *months*, int *days*)**
- **Period of(Days|Weeks|Months|Years)(int *number*)**

  Возвращают период в виде объекта типа **Period** по заданному количеству в указанных единицах измерения времени.
- **int get(Days|Months|Years)()**

  Возвращает количество дней, месяцев или лет, составляющих данный период времени.
- **Period (plus|minus)(Days|Months|Years)(long *number*)**

  Возвращает местную дату, получаемую сложением или вычитанием заданного количества в указанных единицах измерения времени.
- **Period plus(TemporalAmount *amountToAdd*)**
- **Period minus(TemporalAmount *amountToSubtract*)**

  Возвращают момент времени, отстоящий на заданную величину от текущего момента времени. Интерфейс **TemporalAmount** реализуется в классах **Duration** и **Period**.
- **Period with(Days|Months|Years)(int *number*)**

  Возвращает новый период в виде объекта типа **Period** по указанному количеству дней, месяцев или лет.

6.3. Корректоры дат

Для целей планирования нередко требуется рассчитать такие даты, как первый вторник каждого месяца. В классе `TemporalAdjusters` предоставляется целый ряд статических методов для общих видов коррекции дат. Результат выполнения метода коррекции дат передается методу `with()`. Например, первый вторник месяца может быть рассчитан следующим образом:

```
LocalDate firstTuesday = LocalDate.of(year, month, 1)
    .with(TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));
```

Как всегда, метод `with()` возвращает новый объект типа `LocalDate`, не изменяя оригинал. Доступные корректоры дат перечислены в конце этого раздела.

Имеется также возможность создать свой корректор дат, реализовав интерфейс `TemporalAdjuster`. В качестве примера ниже приведен корректор дат, предназначенный для расчета следующего дня недели.

```
TemporalAdjuster NEXT_WORKDAY = w -> {
  var result = (LocalDate) w;
  do {
    result = result.plusDays(1);
  } while (result.getDayOfWeek().getValue() >= 6);
  return result;
};

LocalDate backToWork = today.with(NEXT_WORKDAY);
```

Обратите внимание на то, что параметр лямбда-выражения относится к типу `Temporal` и поэтому должен быть приведен к типу `LocalDate`. Избежать этого

приведения типов можно с помощью метода `ofDateAdjuster()`, ожидающего в качестве параметра лямбда-выражение типа `UnaryOperator<LocalDate>`, как показано ниже.

```
TemporalAdjuster NEXT_WORKDAY = TemporalAdjusters.ofDateAdjuster(w -> {
  LocalDate result = w; // Без приведения типов
  do {
    result = result.plusDays(1);
  } while (result.getDayOfWeek().getValue() >= 6);
  return result;
});
```

java.time.LocalDate 9

- **LocalDate with(TemporalAdjuster *adjuster*)**
  Возвращает результат коррекции текущей данных с помощью заданного корректора.

java.time.temporal.TemporalAdjusters 9

- **static TemporalAdjuster next(DayOfWeek *dayOfWeek*)**
- **static TemporalAdjuster nextOrSame(DayOfWeek *dayOfWeek*)**
- **static TemporalAdjuster previous(DayOfWeek *dayOfWeek*)**
- **static TemporalAdjuster previousOrSame(DayOfWeek *dayOfWeek*)**
  Возвращают корректор даты по указанному дню недели.
- **static TemporalAdjuster dayOfWeekInMonth(int *n*, DayOfWeek *dayOfWeek*)**
- **static TemporalAdjuster lastInMonth(DayOfWeek *dayOfWeek*)**
  Возвращают корректор даты по указанному *n*-му или последнему дню недели в месяце.
- **static TemporalAdjuster firstDayOfMonth()**
- **static TemporalAdjuster firstDayOfNextMonth()**
- **static TemporalAdjuster firstDayOfYear()**
- **static TemporalAdjuster firstDayOfNextYear()**
- **static TemporalAdjuster lastDayOfMonth()**
- **static TemporalAdjuster lastDayOfYear()**
  Возвращают корректор даты по указанному дню месяца или года.

6.4. Местное время

Класс `LocalTime` представляет местное время суток, например 15:30:00. Экземпляр класса `LocalTime` можно получить с помощью метода `now()` или `of()` следующим образом:

```
LocalTime rightNow = LocalTime.now();
LocalTime bedtime = LocalTime.of(22, 30);
    // или LocalTime.of(22, 30, 0)
```

В конце этого раздела перечислены методы, выполняющие общие операции с местным временем. В частности, методы `plus()` и `minus()` заключают в себе операции с местным временем в течение всех суток, как показано в следующей строке кода:

```
// Подъем в 6:30:00!
LocalTime wakeup = bedtime.plusHours(8);
```

НА ЗАМЕТКУ! В самом классе **LocalTime** местное время до и после полудня не разграничивается, поскольку эта обязанность возлагается на средство форматирования, как поясняется далее, в разделе 6.6.

Имеется также класс `LocalDateTime`, представляющий дату и время. Этот класс пригоден для хранения моментов времени в фиксированном часовом поясе, например, для планирования занятий или событий. Но если требуется произвести расчеты с учетом перехода на летнее время или поддерживать пользователей из разных часовых поясов, то следует воспользоваться классом `ZonedDateTime`, рассматриваемым в следующем разделе.

java.time.LocalTime 8

- **static LocalTime now()**
  Возвращает текущее местное время в виде объекта типа **LocalTime**.
- **static LocalTime of(int *hour*, int *minute*)**
- **static LocalTime of(int *hour*, int *minute*, int *second*)**
- **static LocalTime of(int *hour*, int *minute*, int *second*, int *nanoOfSecond*)**
  Возвращают местное время по заданным часам (в пределах от **0** до **23**), минутам и секундам (в пределах от **0** до **59**), а также наносекундам (в пределах от **0** до **999999999**).
- **LocalTime (plus|minus)(Hours|Minutes|Seconds|Nanos)(long *number*)**
  Возвращает местное время в виде объекта типа **LocalTime**, получаемое сложением или вычитанием заданного количества в указанных единицах измерения времени.
- **LocalTime plus(TemporalAmount *amountToAdd*)**
- **LocalTime minus(TemporalAmount *amountToSubtract*)**
  Возвращают момент времени, отстоящий на заданную величину от текущего момента времени.
- **LocalTime with(Hour|Minute|Second|Nano)(int *value*)**
  Возвращает новое местное время в виде объекта типа **LocalTime** по указанному количеству часов, минут, секунд или наносекунд.
- **int getHour()**
  Возвращает количество часов (в пределах от **0** до **23**).
- **int getMinute()**
- **int getSecond()**
  Возвращает количество минут и секунд (в пределах от **0** до **59**).
- **int getNano()**
  Возвращает количество наносекунд (в пределах от **0** до **999999999**).

java.time.LocalTime 8 *(окончание)*

- **int toSecondOfDay()**
- **long toNanoOfDay()**

  Возвращают количество секунд или наносекунд после полуночи.
- **boolean isBefore(LocalTime *other*)**
- **boolean isAfter(LocalTime *other*)**

  Возвращают логическое значение **true**, если текущая дата предшествует указанной дате или следует после нее.

6.5. Поясное время

Еще большую путаницу, чем неравномерность вращения Земли, в расчеты времени вносят часовые пояса, вероятно, потому, что они являются полностью изобретением человечества. Люди во всем мире ведут отсчет от времени по Гринвичу, поэтому одни обедают в 2:00, а другие в 22:00, исходя из потребностей своего желудка, а не времени суток. Так, в Китае обедают в разное время, поскольку на эту страну приходятся четыре условных часовых пояса. Эти пояса считаются условными, поскольку их границы неточны и смещаются, а переход на летнее и зимнее время еще больше усугубляет положение.

Как бы часовые пояса ни досаждали просвещенным, они являются непреложным фактом нашей жизни. Разрабатывая календарное приложение, следует учитывать интересы людей, перемещающихся из одной страны в другую. Так, если требуется вовремя прибыть к 10:00 на конференцию в Нью-Йорке из Берлина, необходимо правильно учесть местное время, определяя время отъезда.

В организации IANA (Internet Assigned Numbers Authority — Комитет по цифровым адресам в Интернете) ведется база данных всех известных в мире часовых поясов (`https://www.iana.org/time-zones`), обновляемая несколько раз в год. Большая часть обновлений относится к изменению правил перехода на летнее и зимнее время. База данных IANA применяется и в Java.

Каждому часовому поясу присваивается свой идентификатор, например `America/New_York` или `Europe/Berlin`. Чтобы выяснить все имеющиеся часовые пояса, достаточно вызвать метод `ZoneId.getAvailableIds()`. На момент написания данной книги насчитывалось почти 600 идентификаторов часовых поясов.

Получая в качестве параметра идентификатор часового пояса, статический метод `ZoneId.of(id)` возвращает объект типа `ZoneId`. Этот объект можно использовать для преобразования объекта типа `LocalDateTime` в объект типа `ZonedDateTime`, вызвав метод `local.atZone(zoneId)`, или для построения объекта типа `ZonedDateTime`, вызвав статический метод `ZonedDateTime.of(year, month, day, hour, minute, second, nano, zoneId)`, как показано в следующем примере кода:

```
// 1969-07-16T09:32-04:00[America/New_York]:
ZonedDateTime apollo11launch = ZonedDateTime.of(1969, 7,
        16, 9, 32, 0, 0, ZoneId.of("America/New_York"));
```

Это конкретный момент времени. Чтобы получить объект типа `Instant`, достаточно сделать вызов `apollo11launch.toInstant()`. С другой стороны, если имеется конкретный момент времени, достаточно сделать вызов `instant.atZone(ZoneId.of("UTC"))`, чтобы получить объект типа `ZonedDateTime`, определяющий поясное время и дату по Гринвичу, или воспользоваться другим объектом типа `ZoneId`, чтобы получить дату и время в любом другом месте планеты.

НА ЗАМЕТКУ! Сокращение UTC обозначает Всеобщее скоординированное время. Это сокращение выбрано в качестве компромисса между английским (Coordinated Universal Time) и французским (Temps Universel Coordiné) обозначениями Всеобщего скоординированного времени, хотя оно неверно на обоих языках. Понятие UTC определяет время по Гринвичу без учета перехода на летнее или зимнее время.

Многие методы из класса `ZonedDateTime` похожи на методы из класса `LocalDateTime` (см. их краткое описание в конце этого раздела). Большинство из них довольно просты, но переход на летнее и зимнее время несколько усложняет расчет поясного времени.

При переходе на летнее время стрелки часов переводятся на один час вперед. Что же произойдет, если рассчитать время, приходящееся на пропущенный час? Например, в 2013 году переход на летнее время в Центральной Европе произошел 31 марта в 2:00. Если попытаться рассчитать время в несуществующий момент 2:30 31 марта 2013 года, то на самом деле будет получено время 3:30.

```
ZonedDateTime skipped = ZonedDateTime.of(
  LocalDate.of(2013, 3, 31),
  LocalTime.of(2, 30),
  ZoneId.of("Europe/Berlin"));
    // Получается время 3:30 31 марта
```

С другой стороны, при переходе на зимнее время стрелки часов переводятся на один час назад, и в одно и то же местное время возникают два момента! При расчете времени в этом промежутке получается более ранний из этих двух моментов времени, как показано ниже.

```
ZonedDateTime ambiguous = ZonedDateTime.of(
    LocalDate.of(2013, 10, 27), // Переход на зимнее время
    LocalTime.of(2, 30),
    ZoneId.of("Europe/Berlin"));
      // 2013-10-27T02:30+02:00[Europe/Berlin]
ZonedDateTime anHourLater = ambiguous.plusHours(1);
      // 2013-10-27T02:30+01:00[Europe/Berlin]
```

Час спустя время будет показывать те же самые часы и минуты, но часовой пояс уже будет смещен. Следует также уделить внимание коррекции даты при пересечении границ перехода на летнее и зимнее время. Так, если требуется назначить совещание на следующей неделе, то не следует вводить промежуток в семь дней, как демонстрируется в приведенном ниже примере кода.

```
// Внимание! Этот код действует неверно при
// переходе на летнее время:
ZonedDateTime nextMeeting =
         meeting.plus(Duration.ofDays(7));
```

Вместо этого лучше воспользоваться классом `Period` следующим образом:

```
ZonedDateTime nextMeeting =
    meeting.plus(Period.ofDays(7)); // Верно!
```

ВНИМАНИЕ! Имеется также класс **`OffsetDateTime`**, представляющий время со смещением относительно времени UTC, но без учета правил смены часовых поясов. Этот класс предназначен для специального применения, требующего, в частности, отсутствия этих правил, в том числе и некоторых сетевых протоколов. Для получения поясного времени в удобочитаемом формате лучше пользоваться классом **`ZonedDateTime`**.

Применение класса `ZonedDateTime` демонстрируется в примере программы из листинга 6.3.

Листинг 6.3. Исходный код из файла **`zonedtimes/ZonedTimes.java`**

```
1  package zonedtimes;
2
3  /**
4   * @version 1.0 2016-05-10
5   * @author Cay Horstmann
6   */
7  import java.time.*;
8
9  public class ZonedTimes
10 {
11   public static void main(String[] args)
12   {
13     ZonedDateTime apollo11launch = ZonedDateTime.of(
14                  1969, 7, 16, 9, 32, 0, 0,
15                  ZoneId.of("America/New_York"));
16           // 1969-07-16T09:32-04:00[America/New_York]
17     System.out.println("apollo11launch: "
18                     + apollo11launch);
19
20     Instant instant = apollo11launch.toInstant();
21     System.out.println("instant: " + instant);
22
23     ZonedDateTime zonedDateTime =
24                  instant.atZone(ZoneId.of("UTC"));
25     System.out.println("zonedDateTime: "
26                     + zonedDateTime);
27
28     ZonedDateTime skipped = ZonedDateTime.of(
29                  LocalDate.of(2013, 3, 31),
30                  LocalTime.of(2, 30),
31                  ZoneId.of("Europe/Berlin"));
32              // Формирование даты 31 марта 3:30
34     System.out.println("skipped: " + skipped);
35
36     ZonedDateTime ambiguous = ZonedDateTime.of(
37                  LocalDate.of(2013, 10, 27),
38               // Переход на зимнее время
39                  LocalTime.of(2, 30),
40                  ZoneId.of("Europe/Berlin"));
```

```
41              // 2013-10-27T02:30+02:00[Europe/Berlin]
42      ZonedDateTime anHourLater = ambiguous.plusHours(1);
43         // 2013-10-27T02:30+01:00[Europe/Berlin]
44      System.out.println("ambiguous: " + ambiguous);
45      System.out.println("anHourLater: " + anHourLater);
46
47      ZonedDateTime meeting = ZonedDateTime.of(
48                      LocalDate.of(2013, 10, 31),
49                      LocalTime.of(14, 30),
50                      ZoneId.of("America/Los_Angeles"));
51      System.out.println("meeting: " + meeting);
52      // Внимание! Не годится, так как не учитывает
53      // переход с летнего времени на зимнее:
54      ZonedDateTime nextMeeting =
55                      meeting.plus(Duration.ofDays(7));
56      System.out.println("nextMeeting: " + nextMeeting);
57      nextMeeting =
58          meeting.plus(Period.ofDays(7)); // Верно!
59      System.out.println("nextMeeting: " + nextMeeting);
60    }
61 }
```

java.time.ZonedDateTime 8

- **static ZonedDateTime now()**

  Возвращает поясное время и дату в виде объекта типа **ZonedDateTime**.
- **static ZonedDateTime of(int *year*, int *month*, int *dayOfMonth*, int *hour*, int *minute*, int *second*, int *nanoOfSecond*, ZoneId *zone*)**
- **static ZonedDateTime of(LocalDate *date*, LocalTime *time*, ZoneId *zone*)**
- **static ZonedDateTime of(LocalDateTime *localDateTime*, ZoneId *zone*)**
- **static ZonedDateTime ofInstant(Instant *instant*, ZoneId *zone*)**

  Возвращают поясное время и дату в виде объекта типа **ZonedDateTime** по заданным параметрам и часовому поясу.
- **ZonedDateTime (plus|minus)(Days|Weeks|Months|Years|Hours|Minutes|Seconds|Nanos)(long *number*)**

  Возвращает поясное время и дату в виде объекта типа **ZonedDateTime**, получаемые сложением или вычитанием заданного количества в указанных единицах измерения времени.
- **ZonedDateTime plus(TemporalAmount *amountToAdd*)**
- **ZonedDateTime minus(TemporalAmount *amountToSubtract*)**

  Возвращают момент времени, отстоящий на заданную величину от текущего момента времени.
- **ZonedDateTime with(DayOfMonth|DayOfYear|Month|Year|Hour|Minute|Second|Nano)(int *value*)**

  Возвращает новое поясное время и дату в виде объекта типа **ZonedDateTime** по заданной величине в указанных единицах измерения времени.
- **ZonedDateTime withZoneSameInstant(ZoneId *zone*)**
- **ZonedDateTime withZoneSameLocal(ZoneId *zone*)**

  Возвращают новое поясное время и дату в виде объекта типа **ZonedDateTime** в указанном часовом поясе, представляя тот же самый момент времени или местное время.

java.time.ZonedDateTime 8 *(окончание)*

- **int getDayOfMonth()**

  Возвращает день месяца (в пределах от **1** до **31**).
- **int getDayOfYear()**

  Возвращает день года (в пределах от **1** до **366**).
- **DayOfWeek getDayOfWeek()**

  Возвращает день недели в виде значения из перечисления **DayOfWeek**.
- **Month getMonth()**
- **int getMonthValue()**

  Возвращают месяц в виде значения из перечисления **Month** или числа в пределах от **1** до **12**.
- **int getYear()**

  Возвращает год (в пределах от **-999999999** до **999999999**).
- **int getHour()**

  Возвращает количество часов (в пределах от **0** до **23**).
- **int getMinute()**
- **int getSecond()**

  Возвращают количество минут и секунд (в пределах от **0** до **59**).
- **int getNano()**
- Возвращает количество наносекунд (в пределах от **0** до **999999999**).
- **public ZoneOffset getOffset()**

  Возвращает смещение относительно времени UTC. Смещение может изменяться в пределах от **-12:00** до **+14:00**. У некоторых часовых поясов может быть дробное смещение. При переходе на летнее или зимнее время смещение изменяется.
- **LocalDate toLocalDate()**
- **LocalTime toLocalTime()**
- **LocalDateTime toLocalDateTime()**
- **Instant toInstant()**

  Возвращают местную дату, время, дату и время или соответствующий момент времени.
- **boolean isBefore(ChronoZonedDateTime *other*)**
- **boolean isAfter(ChronoZonedDateTime *other*)**

  Возвращают логическое значение **true**, если текущее поясное время и дата предшествует указанному поясному времени и дате или следует после них.

6.6. Форматирование и синтаксический анализ даты и времени

В классе `DateTimeFormatter` предоставляются перечисленные ниже средства форматирования для вывода значений даты и времени.

- Предопределенные стандартные средства форматирования, перечисленные в табл. 6.1.

- Средства форматирования с учетом региональных настроек.
- Средства форматирования по специальным шаблонам.

Чтобы воспользоваться одним из стандартных средств форматирования даты и времени, достаточно вызвать его метод `format()` следующим образом:

```
String formatted = DateTimeFormatter.ISO_DATE_TIME
                                    .format(apollo11launch);
   // 1969-07-16T09:32:00-05:00[America/New_York]
```

Таблица 6.1. Предопределенные средства форматирования

Средство форматирования	Описание	Пример
`BASIC_ISO_DATE`	Год, месяц, день, смещение часового пояса без разделителей	`19690716-0500`
`ISO_LOCAL_DATE`, `ISO_LOCAL_TIME`, `ISO_LOCAL_DATE_TIME`	Разделители `-`, `:`, `T`	`1969-07-16`, `09:32:00`, `1969-07-16T09:32:00`
`ISO_OFFSET_DATE`, `ISO_OFFSET_TIME`, `ISO_OFFSET_DATE_TIME`	Аналогично `ISO_LOCAL_XXX`, но со смещением часового пояса	`1969-07-16-05:00`, `09:32:00-05:00`, `1969-07-16T09:32:00-05:00`
`ISO_ZONED_DATE_TIME`	Со смещением часового пояса и идентификатором часового пояса	`1969-07-16T09:32:00-05:00 [America/New_York]`
`ISO_INSTANT`	Время в формате UTC, где `Z` обозначает идентификатор часового пояса	`1969-07-16T14:32:00Z`
`ISO_DATE`, `ISO_TIME`, `ISO_DATE_TIME`	Аналогично `ISO_OFFSET_DATE`, `ISO_OFFSET_TIME` и `ISO_ZONED_DATE_TIME`, но сведения о часовом поясе указываются дополнительно, хотя и не обязательно	`1969-07-16-05:00`, `09:32:00-05:00`, `1969-07-16T09:32:00-05:00[America/New_York]`
`ISO_ORDINAL_DATE`	Год и день года для местной даты типа `LocalDate`	`1969-197`
`ISO_WEEK_DATE`	Год, неделя и день недели для местной даты типа `LocalDate`	`1969-W29-3`
`RFC_1123_DATE_TIME`	Стандарт для отметок времени в электронной почте, кодируемых по стандарту RFC 822 и обновляемых четырьмя цифрами для обозначения года по стандарту RFC 1123	`Wed, 16 Jul 1969 09:32:00 -0500`

Стандартные средства предназначены в основном для форматирования машинно-читаемых отметок времени. Для представления дат и времени в удобочитаемом виде служат средства форматирования с учетом региональных настроек. Они поддерживают четыре стиля форматирования даты и времени: `SHORT`, `MEDIUM`, `LONG` и `FULL` (табл. 6.2).

Таблица 6.2. Стили форматирования с учетом региональных настроек

Стиль	Дата	Время
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT
FULL	Wednesday, July 16, 1969	9:32:00 AM EDT

Для создания средства форматирования с учетом региональных настроек служат статические методы `ofLocalizedDate()`, `ofLocalizedTime()` и `ofLocalizedDateTime()`. Ниже приведен характерный тому пример.

```
DateTimeFormatter formatter = DateTimeFormatter
                .ofLocalizedDateTime(FormatStyle.LONG);
String formatted = formatter.format(apollo11launch);
    // July 16, 1969 9:32:00 AM EDT
```

В этих методах применяются региональные настройки, выбираемые по умолчанию. Чтобы выбрать другие региональные настройки, достаточно вызвать метод `withLocale()` следующим образом:

```
formatted = formatter.withLocale(Locale.FRENCH)
                    .format(apollo11launch);
    // 16 juillet 1969 09:32:00 EDT
```

У перечислений `DayOfWeek` и `Month` имеются методы `getDisplayName()` для получения наименований дней недели и месяца в разных форматах и региональных настройках, как показано ниже. Подробнее о региональных настройках речь пойдет в главе 7.

```
for (DayOfWeek w : DayOfWeek.values())
  // вывести дни недели Mon Tue Wed Thu Fri Sat Sun
  // на английском языке:
  System.out.print(w.getDisplayName(TextStyle.SHORT,
                                    Locale.ENGLISH) + " ");
```

НА ЗАМЕТКУ Класс **`java.time.format.DateTimeFormatter`** служит для замены класса **`java.util.DateFormat`**. Если же требуется получить экземпляр последнего ради обратной совместимости, следует вызвать метод **`formatter.toFormat()`**.

Наконец, можно создать свой формат даты, указав шаблон. Так, в следующей строке кода:

```
formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
```

дата форматируется в виде `Wed 1969-07-16 09:32`. Каждая буква в шаблоне обозначает отдельное поле даты и времени, а количество повторений букв — конкретный формат, выбираемый по правилам, которые не совсем ясны и, по-видимому, органично выработались со временем. Наиболее употребительные элементы шаблонов для форматирования даты и времени перечислены в табл. 6.3.

Таблица 6.3. Наиболее употребительные знаки в шаблонах для форматирования даты и времени

Константа перечислимого типа `ChronoField` или назначение	Перевод	Примеры
`ERA`	Эра	`G: AD, GGGG: Anno Domini, GGGGG: A`
`YEAR_OF_ERA`	Год эры	`yy: 69, yyyy: 1969`
`MONTH_OF_YEAR`	Месяц года	`M: 7, MM: 07, MMM: Jul, MMMM: July, MMMMM: J`
`DAY_OF_MONTH`	День месяца	`d: 6, dd: 06`
`DAY_OF_WEEK`	День недели	`e: 3, E: Wed, EEEE: Wednesday, EEEEE: W`
`HOUR_OF_DAY`	Час дня	`H: 9, HH: 09`
`CLOCK_HOUR_OF_AM_PM`	Полный час дня по часам до или после полудня	`K: 9, KK: 09`
`AMPM_OF_DAY`	Время суток до или после полудня	`a: AM`
`MINUTE_OF_HOUR`	Минута часа	`mm: 02`
`SECOND_OF_MINUTE`	Секунда минуты	`ss: 00`
`NANO_OF_SECOND`	Наносекунда секунды	`nnnnnn: 000000`
Идентификатор часового пояса		`VV: America/New_York`
Наименование часового пояса		`z: EDT, zzzz: Eastern Daylight Time`
Смещение часового пояса		`x: -04, xx: -0400, xxx: -04:00, XXX:` то же самое, но `Z` обозначает нуль
Локализованное смещение часового пояса		`O: GMT-4, OOOO: GMT-04:00`
Модифицированный день по юлианскому календарю		`g: 58243`

Для синтаксического анализа значения даты и времени из символьной строки служит статический метод `parse()`, как показано ниже. В первом вызове этого метода используется стандартное средство форматирования `ISO_LOCAL_DATE`, а во втором вызове — специальное средство форматирования.

```
LocalDate churchsBirthday = LocalDate.parse("1903-06-14");
ZonedDateTime apollo11launch =
     ZonedDateTime.parse("1969-07-16 03:32:00-0400",
     DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));
```

Порядок форматирования дат и времени демонстрируется в примере программы из листинга 6.4.

Листинг 6.4. Исходный код из файла **`formatting/Formatting.java`**

```
1  package formatting;
2
3  /**
```

```
4    * @version 1.0 2016-05-10
5    * @author Cay Horstmann
6    */
7  import java.time.*;
8  import java.time.format.*;
9  import java.util.*;
10
11 public class Formatting
12 {
13   public static void main(String[] args)
14   {
15     ZonedDateTime apollo11launch = ZonedDateTime.of(
16                       1969, 7, 16, 9, 32, 0, 0,
17                       ZoneId.of("America/New_York"));
18
19     String formatted = DateTimeFormatter
20           .ISO_OFFSET_DATE_TIME.format(apollo11launch);
21        // 1969-07-16T09:32:00-04:00
22     System.out.println(formatted);
23
24     DateTimeFormatter formatter = DateTimeFormatter
25           .ofLocalizedDateTime(FormatStyle.LONG);
26     formatted = formatter.format(apollo11launch);
27        // July 16, 1969 9:32:00 AM EDT
28     System.out.println(formatted);
29     formatted = formatter.withLocale(Locale.FRENCH)
30                 .format(apollo11launch);
31        // 16 juillet 1969 09:32:00 EDT
32     System.out.println(formatted);
33
34     formatter = DateTimeFormatter
35                 .ofPattern("E yyyy-MM-dd HH:mm");
36     formatted = formatter.format(apollo11launch);
37     System.out.println(formatted);
38
39     LocalDate churchsBirthday =
40               LocalDate.parse("1903-06-14");
41     System.out.println("churchsBirthday: "
42                        + churchsBirthday);
43     apollo11launch = ZonedDateTime.parse(
44                    "1969-07-16 03:32:00-0400",
45                    DateTimeFormatter
46                    .ofPattern("yyyy-MM-dd HH:mm:ssxx"));
47     System.out.println("apollo11launch: "
48                        + apollo11launch);
49
50     for (DayOfWeek w : DayOfWeek.values())
51       System.out.print(w.getDisplayName(TextStyle.SHORT,
52                        Locale.ENGLISH) + " ");
53   }
54 }
```

java.time.format.DateTimeFormatter 8

- **String format(TemporalAccessor *temporal*)**

  Форматирует заданное значение. Интерфейс **TemporalAccessor** реализуется в классах **Instant**, **LocalDate**, **LocalTime**, **LocalDateTime** и **ZonedDateTime**, а также во многих других классах.
- **static DateTimeFormatter ofLocalizedDate(FormatStyle *dateStyle*)**
- **static DateTimeFormatter ofLocalizedTime(FormatStyle *timeStyle*)**
- **static DateTimeFormatter ofLocalizedDateTime(FormatStyle *dateTimeStyle*)**
- **static DateTimeFormatter ofLocalizedDateTime(FormatStyle *dateStyle*, FormatStyle *timeStyle*)**

  Возвращают средство форматирования по указанным стилям, которые определяются в виде значений **SHORT**, **MEDIUM**, **LONG** и **FULL** в перечислении **FormatStyle**.
- **DateTimeFormatter withLocale(Locale *locale*)**

  Возвращает средство форматирования, равнозначное данному средству форматирования, используя указанные региональные настройки.
- **static DateTimeFormatter ofPattern(String *pattern*)**
- **static DateTimeFormatter ofPattern(String *pattern*, Locale *locale*)**

  Возвращают средство форматирования, используя указанный шаблон и региональные настройки. Синтаксис шаблонов для форматирования даты и времени см. в табл. 6.3.

java.time.LocalDate 8

- **static LocalDate parse(CharSequence *text*)**
- **static LocalDate parse(CharSequence *text*, DateTimeFormatter *formatter*)**

  Возвращают местную дату в виде объекта типа **LocalDate**, используя выбираемое по умолчанию или заданное средство форматирования.

java.time.ZonedDateTime 8

- **static ZonedDateTime parse(CharSequence *text*)**
- **static ZonedDateTime parse(CharSequence *text*, DateTimeFormatter *formatter*)**

  Возвращают поясное время и дату в виде объекта типа **ZonedDateTime**, используя выбираемое по умолчанию или заданное средство форматирования.

6.7. Взаимодействие с унаследованным кодом

Новый прикладной интерфейс Java API даты и времени должен обеспечивать нормальное взаимодействие с уже имеющимися классами. К их

числу относятся широко распространенные классы `java.util.Date`, `java.util.GregorianCalendar` и `java.sql.Date/Time/Timestamp`.

Класс `Instant` очень похож на класс `java.util.Date`. В версии Java 8 этот класс был дополнен двумя методами. В частности, метод `toInstant()` служит для преобразования типа `Date` в тип `Instant`, а статический метод `from()` — для обратного преобразования этих типов даты и времени.

Аналогично класс `ZonedDateTime` очень похож на класс `java.util.GregorianCalendar` и дополнен соответствующими методами преобразования в версии Java 8. В частности, метод `toZonedDateTime()` служит для преобразования типа `GregorianCalendar` в тип `ZonedDateTime`, а метод `from()` — для обратного преобразования этих типов даты и времени.

Еще один ряд преобразований предусмотрен для классов даты и времени из пакета `java.sql`. Кроме того, средство форматирования типа `DateTimeFormatter` можно передать унаследованному коду, где применяется класс `java.text.Format`. Все эти методы преобразования сведены в табл. 6.4.

Таблица 6.4. Методы взаимного преобразования классов из пакета `java.time` и унаследованного кода

Классы	Метод преобразования в унаследованный код	Метод преобразования из унаследованного кода
`Instant` ↔ `java.util.GregorianCalendar`	`Date.from(instant)`	`date.toInstant()`
`ZonedDateTime` ↔ `java.util.GregorianCalendar`	`GregorianCalendar.from(zonedDateTime)`	`cal.toZonedDateTime()`
`Instant` ↔ `java.sql.Timestamp`	`TimeStamp.from(instant)`	`timestamp.toInstant()`
`LocalDateTime` ↔ `java.sql.Timestamp`	`Timestamp.valueOf(localDateTime)`	`timeStamp.toLocalDateTime()`
`LocalDate` ↔ `java.sql.Date`	`Date.valueOf(localDate)`	`date.toLocalDate()`
`LocalTime` ↔ `java.sql.Time`	`Time.valueOf(localTime)`	`time.toLocalTime()`
`DateTimeFormatter` → `java.text.DateFormat`	`formatter.toFormat()`	Отсутствует
`java.util.TimeZone` ↔ `ZoneId`	`Timezone.getTimeZone(id)`	`timeZone.toZoneId()`
`java.nio.file.attribute.FileTime` ↔ `Instant`	`FileTime.from(instant)`	`fileTime.toInstant()`

Из этой главы вы узнали, как пользоваться библиотекой даты и времени, внедренной в версии Java 8, чтобы оперировать значениями даты и времени повсюду в мире. В следующей главе речь пойдет об интернационализации прикладных программ на Java. В ней будет показано, как форматировать сообщения, выводимые программой, числа и денежные суммы в привычном для пользователей виде, где бы они ни находились.

ГЛАВА 7

Интернационализация

В этой главе...

- Региональные настройки
- Форматирование чисел
- Форматирование даты и времени
- Сортировка и нормализация
- Форматирование сообщений
- Ввод-вывод текста
- Комплекты ресурсов
- Пример интернационализации прикладной программы

Создавая приложение, разработчики надеются, что их программным продуктом заинтересуются многие пользователи. К тому же благодаря Интернету преодолеваются границы между разными странами. С другой стороны, если разработчики не принимают во внимание международных пользователей, то сами создают искусственные препятствия для широкого распространения своих программных продуктов по всему миру.

Java стал первым языком программирования, в котором изначально были предусмотрены средства интернационализации. Строки в Java формируются из символов в Юникоде (Unicode). Поддержка этого стандарта кодирования символов позволяет разрабатывать программы на Java, способные обрабатывать тексты на любом из языков, существующих в мире.

Многие программисты считают, что для интернационализации своих приложений им достаточно воспользоваться Юникодом и перевести на нужный язык все сообщения пользовательского интерфейса. Но этого явно недостаточно, потому что интернационализация программы означает нечто большее, чем поддержка кодировки символов в Юникоде. Дата, время, денежные суммы и даже

числа могут по-разному представляться на различных языках. Необходимо также найти простой способ настройки команд меню, надписей на кнопках, сообщений и комбинаций клавиш на выбранный язык и региональный стандарт.

В этой главе рассматриваются способы интернационализации прикладных программ на Java, а также особенности представления времени, даты, чисел, текста и элементов графического пользовательского интерфейса с учетом региональных стандартов. Кроме того, в ней обсуждаются некоторые инструментальные средства, предназначенные для интернационализации программ. В конце главы в качестве примера будет рассмотрена программа калькуляции пенсионных сбережений с пользовательским интерфейсом на английском, немецком и китайском языках.

7.1. Региональные настройки

Приложение, которое адаптировано для международного рынка, легко определить по возможности выбора языка, используемого для работы с ним. Но профессионально адаптированные приложения могут иметь разные региональные настройки даже для тех стран, в которых используется одинаковый язык. Эту ситуацию очень точно подметил некогда Оскар Уайльд: "Теперь у нас, действительно, все, как в Америке, естественно, кроме языка".

7.1.1. Назначение региональных настроек

Если предоставляются международные версии прикладной программы, все ее сообщения должны быть переведены на местный язык. Но одного лишь перевода текста пользовательского интерфейса явно недостаточно. Существует еще много других, более тонких различий. Например, в Англии и Германии для представления десятичных чисел применяются разные форматы. Число, понятное англичанам в формате `123,456.78`, должно быть отображено в формате `123.456,78` для пользователей из Германии. Иными словами, точка и запятая в качестве разделителя дробной части и разделителя групп *по-разному* используются в этих странах!

Похожие различия можно заметить и в способах представления даты. В США привыкли отображать даты в формате месяц/день/год, в Германии используют более практичный порядок — день/месяц/год, а в Китае все наоборот — год/месяц/день. Таким образом, дата `3/22/61` должна быть представлена в формате `22.03.1961` для немецкого пользователя. Если названия месяцев написаны полностью, то разница в способах представления дат становится еще более очевидной. Например, дата `March 22, 1961`, понятная для американского пользователя, должна быть представлена как `22. März 1961` для немецкого пользователя.

Такие местные предпочтения пользователей фиксируются в *региональных настройках*. Всякий раз, когда требуется представить числа, даты, денежные суммы и прочие элементы, формирование которых отличается по языкам и странам мира, следует пользоваться прикладными интерфейсами API, в которых учитываются региональные настройки.

7.1.2. Указание региональных настроек

Региональные настройки содержат следующие составляющие.

1. Язык, обозначаемый двумя или тремя строчными буквами, например **en** (английский), **de** (немецкий) или **zh** (китайский). Наиболее употребительные коды языков перечислены в табл. 7.1.
2. Дополнительно письмо, обозначаемое четырьмя буквами, первая из которых является прописной, например **Latn** (латынь), **Cyrl** (кириллица) или **Hant** (традиционные китайские иероглифы). Это удобно, поскольку в ряде языков (например, в сербском) употребляется как латынь, так и кириллица, а некоторые китайские пользователи предпочитают традиционные иероглифы упрощенным.
3. Дополнительно страна или регион, обозначаемые двумя прописными буквами или тремя цифрами, например **US** (Соединенные Штаты) или **CH** (Швейцария). Наиболее употребительные коды стран перечислены в табл. 7.2.
4. Дополнительно *вариант*, обозначающий различные свойства языка, в том числе диалекты или правила произношения. В настоящее время варианты употребляются редко. Раньше употреблялся "новонорвежский" вариант норвежского языка, но теперь он выражается отдельным кодом языка **nn**. Прежние варианты японского императорского календаря и тайских цифр теперь выражаются как расширения, поясняемые ниже.
5. Дополнительно расширение. Расширения описывают локальные установки для календарей (например, японского), чисел (тайских цифр вместо арабских) и т.д. Некоторые из этих расширений определены в стандарте на Юникод. Расширения начинаются с обозначения **u-** и продолжаются двухбуквенным кодом, указывающим назначение расширения: **ca** — для календаря, **nu** — для чисел и т.д. Например, расширение **u-nu-thai** обозначает употребление тайских цифр. Другие расширения совершенно произвольны и начинаются с обозначения **x-**, например **x-java**.

Правила для региональных настроек сформулированы в памятной записке BCP 47 "Best Current Practices" (Передовые современные методики) Рабочей группы инженерной поддержки Интернета (Internet Engineering Task Force; `http://tools.ietf.org/html/bcp47`). Более доступное краткое изложение этих правил можно найти по адресу `www.w3.org/International/articles/language-tags`.

Таблица 7.1. Наиболее употребительные коды языков по стандарту ISO 639-1

Язык	Код	Язык	Код
Китайский	**zh**	Японский	**ja**
Датский	**da**	Корейский	**ko**
Голландский	**du**	Норвежский	**no**
Английский	**en**	Португальский	**pt**
Французский	**fr**	Испанский	**es**
Финский	**fi**	Шведский	**sv**
Итальянский	**it**	Турецкий	**tr**

Таблица 7.2. Наиболее употребительные коды стран по стандарту ISO 3166-1

Страна	Код	Страна	Код
Австрия	**AT**	Япония	**JP**
Бельгия	**BE**	Корея	**KR**
Канада	**CA**	Нидерланды	**NL**
Китай	**CN**	Норвегия	**NO**
Дания	**DK**	Португалия	**PT**
Финляндия	**FI**	Испания	**ES**
Германия	**DE**	Швеция	**SE**
Великобритания	**GB**	Швейцария	**CH**
Греция	**GR**	Тайвань	**TW**
Ирландия	**IE**	Турция	**TR**
Италия	**IT**	Соединенные Штаты	**US**

Коды языков и стран кажутся на первый взгляд выбранными несколько произвольно, поскольку некоторые из них происходят от местных языков. Так, Deutsch по-немецки означает "немецкий", а zhongwen по-китайски (в латинской транскрипции) — "китайский", отсюда и коды этих языков **de** и **zh** соответственно. Швейцария обозначается кодом **CH**, происходящим от латинского термина *Confoederatio Helvetica*, означающего "Швейцарская конфедерация".

Региональные настройки описываются дескрипторами в виде символьных строк, где элементы региональных настроек указываются через дефис, например **en-US** для США или **de-DE** для Германии. В Швейцарии приняты четыре официальных языка (немецкий, французский, итальянский и ретороманский), поэтому для немецкоязычного пользователя из Швейцарии потребуются региональные настройки **de-CH**, соблюдающие правила немецкого языка, но выражающие денежные суммы в швейцарских франках, а не в евро. Если же указать только язык (**de**), то такие региональные настройки нельзя использовать для выражения характерных для конкретной страны особенностей вроде представления денежных сумм в местной валюте.

Создать объект типа `Locale` из символьной строки дескриптора можно следующим образом:

```
Locale usEnglish = Locale.forLanguageTag("en-US");
```

Метод `toLanguageTag()` возвращает дескриптор языка из заданных региональных настроек. Так, в результате вызова `Locale.US.toLanguageTag()` возвращается символьная строка `"en-US"`.

Ради удобства для различных стран заранее определены следующие объекты региональных настроек:

```
Locale.CANADA
Locale.CANADA_FRENCH
Locale.CHINA
Locale.FRANCE
Locale.GERMANY
Locale.ITALY
Locale.JAPAN
Locale.KOREA
Locale.PRC
```

```
Locale.TAIWAN
Locale.UK
Locale.US
```

Ряд заранее определенных региональных настроек обозначают только язык, но не страну или регион, как показано ниже. Наконец, статический метод `getAvailableLocales()` возвращает массив всех региональных настроек, известных виртуальной машине.

```
Locale.CHINESE
Locale.ENGLISH
Locale.FRENCH
Locale.GERMAN
Locale.ITALIAN
Locale.JAPANESE
Locale.KOREAN
Locale.SIMPLIFIED_CHINESE
Locale.TRADITIONAL_CHINESE
```

НА ЗАМЕТКУ! Все коды языков можно получить, сделав вызов **`Locale.getISOLanguages()`**, а все коды стран, — сделав вызов **`Locale.getISOCountries()`**.

7.1.3. Региональные настройки по умолчанию

Статический метод `getDefault()` из класса `Locale` позволяет определить региональные настройки, которые выбираются в операционной системе по умолчанию. Изменить эти настройки по умолчанию можно, вызвав метод `setDefault()`, но не следует забывать, что действие этого метода распространяется только на прикладную программу на Java, а не на всю операционную систему.

В некоторых операционных системах допускается указывать разные региональные настройки для форматирования и отображения сообщений. Например, для франкоязычного пользователя прикладной программы, проживающего в Соединенных Штатах, может быть предоставлено меню на французском языке, но денежные суммы указаны в долларах США. Чтобы получить предпочтительные для пользователя региональные настройки форматирования и отображения сообщений, достаточно сделать следующие вызовы:

```
Locale displayLocale =
       Locale.getDefault(Locale.Category.DISPLAY);
Locale formatLocale =
       Locale.getDefault(Locale.Category.FORMAT);
```

НА ЗАМЕТКУ! В UNIX можно указать отдельные региональные настройки для форматирования чисел, денежных сумм и дат, установив соответствующим образом переменные окружения **`LC_NUMERIC`**, **`LC_MONETARY`** и **`LC_TIME`**. Хотя в Java они во внимание не принимаются.

СОВЕТ. Для проверки интернационализации своей программы попробуйте сменить региональные настройки по умолчанию. С этой целью укажите язык и страну при запуске программы на выполнение из командной строки. Например, в приведенной ниже команде запуска программы задается немецкий язык для Швейцарии.

```
java -Duser.language=de -Duser.region=CH МояПрограмма
```

7.1.4. Отображаемые имена

Что же можно сделать с полученными региональными настройками? Оказывается, не так уж и много. Единственную пользу можно извлечь из тех методов в классе `Locale`, которые определяют коды языка и страны. Наиболее важным из них является метод `getDisplayName()`, который возвращает отображаемое имя в виде символьной строки, описывающей региональные настройки. Она содержит не какие-то загадочные двухбуквенные коды, а вполне удобочитаемое описание, как демонстрируется в приведенном ниже примере.

```
German (Switzerland)
```

Но дело в том, что это отображаемое имя выводится на языке региональных настроек по умолчанию, что далеко не всегда удобно. Так, если пользователь уже выбрал немецкий язык интерфейса, отображаемое имя региональной настройки следует представить на немецком языке, передав в качестве параметра соответствующие региональные настройки:

```
var loc = new Locale("de", "CH");
System.out.println(loc.getDisplayName(Locale.GERMAN));
```

В результате выполнения этого фрагмента кода отображаемое имя региональных настроек будет выведено на указанном в них языке:

```
Deutsch (Schweiz)
```

Из данного примера становится ясно, зачем нужны объекты типа `Locale`. Передавая их методам, способным реагировать на региональные настройки, можно отображать текст на языке, понятном пользователю. Примеры применения таких объектов рассматриваются далее в этой главе.

ВНИМАНИЕ! Даже такие обыденные операции, как приведение символьной строки к нижнему или верхнему регистру букв, могут быть выполнены с учетом региональных настроек. Например, строчная буква **i** в турецком правописании обозначается без точки как `ı`. В связи с этим прикладные программы, в которых предпринимались попытки нормализовать символьные строки путем их сохранения в нижнем регистре, таинственным образом давали сбои у турецких пользователей. Поэтому в подобных случаях целесообразно пользоваться методами **`toUpperCase()`** и **`toLowerCase()`**, принимающими в качестве аргумента объект типа **`Locale`**, представляющий региональные настройки, как демонстрируется в следующем примере кода:

```
String cmd =
       "QUIT".toLowerCase(Locale.forLanguageTag("tr"));
        // слово "quıt" (выйти) с буквой ı без точки
```

Разумеется, если в результате вызова метода **`Locale.getDefault()`** в прикладной программе, локализованной для турецких пользователей, получить используемые по умолчанию региональные настройки, то вызов `"QUIT".toLowerCase()` не приведет в итоге к слову **`"quit"`**.

Чтобы привести строки на конкретном языке к нижнему регистру, методу **`toLowerCase()`** следует передать соответствующую региональную настройку в качестве параметра.

НА ЗАМЕТКУ! Для выполнения операций ввода-вывода можно задать региональные настройки явным образом, как поясняется ниже.

- При чтении чисел из объекта типа **Scanner** региональные настройки можно задать с помощью метода **useLocale()**.
- Методы **String.format()** и **PrintWriter.printf()** дополнительно принимают в качестве аргумента объект типа **Locale**, представляющий региональные настройки.

java.util.Locale 1.1

- **Locale(String *language*)**
- **Locale(String *language*, String *country*)**
- **Locale(String *language*, String *country*, String *variant*)**

  Создают объект типа **Locale** региональных настроек с учетом указанного языка, страны и варианта языка. Вместо вариантов языка в новом коде рекомендуется использовать языковые дескрипторы по стандарту IETF BCP 47.
- **static Locale forLanguageTag(String *languageTag*)** 7

  Создает объект типа **Locale** региональных настроек по заданному языковому дескриптору.
- **static Locale getDefault()**

  Возвращает региональные настройки, используемые по умолчанию.
- **static void setDefault(Locale *loc*)**

  Задает региональные настройки, используемые по умолчанию.
- **String getDisplayName()**

  Возвращает отображаемое имя текущих региональных настроек на том языке, который указан в текущих региональных настройках.
- **String getDisplayName(Locale *loc*)**

  Возвращает отображаемое имя заданных региональных настроек на том языке, который указан в заданных региональных настройках.
- **String getLanguage()**

  Возвращает код языка (две строчные буквы) по стандарту ISO 639-1.
- **String getDisplayLanguage()**

  Возвращает название языка в формате текущих региональных настроек.
- **String getDisplayLanguage(Locale *loc*)**

  Возвращает название языка в формате заданных региональных настроек.
- **String getCountry()**

  Возвращает код страны (две прописные буквы) по стандарту ISO 3166-1.
- **static String[] getISOCountries()**
- **static Set<String> getISOCountries(Locale.IsoCountryCode *type*)** 9

  Возвращают двух-, трех- или четырехбуквенные коды всех стран. В качестве параметра ***type*** указывается одна из следующих перечислимых констант: **PART1_ALPHA2**, **PART1_ALPHA3** или **PART3**.
- **String getDisplayCountry()**

  Возвращает название страны в формате текущих региональных настроек.

java.util.Locale 1.1 *(окончание)*

- **String getDisplayCountry(Locale *loc*)**

  Возвращает название страны в формате заданных региональных настроек.

- **String toLanguageTag()** 7

  Возвращает языковой дескриптор по стандарту IETF BCP 47 для текущих региональных настроек, например **"de-CH"** (немецкий язык, Швейцария).

- **String toString()**

  Возвращает описание региональных настроек в виде кодов языка и страны, разделенных знаком подчеркивания (например, **"de_CH"**, т.е. немецкий язык, Швейцария). Этим методом рекомендуется пользоваться только для целей отладки.

7.2. Форматирование чисел

Как упоминалось выше, в разных странах и регионах применяются различные способы представления чисел и денежных сумм. В пакете `java.text` содержатся классы, позволяющие форматировать числа и выполнять синтаксический анализ их строкового представления.

7.2.1. Форматирование числовых значений

Чтобы отформатировать число в соответствии с конкретными региональными настройками, необходимо выполнить следующие действия.

1. Получить объект региональных настроек, как пояснялось в предыдущем разделе.
2. Вызвать фабричный метод для получения форматирующего объекта.
3. Применить форматирующий объект для форматирования числа или синтаксического анализа его строкового представления.

В качестве фабричных служат статические методы `getNumberInstance()`, `getCurrencyInstance()` и `getPercentInstance()` из класса `NumberFormat`. Они получают в качестве параметра объект типа `Locale` и возвращают объекты, предназначенные для форматирования чисел, денежных сумм и числовых величин в процентах. Например, для выражения денежной суммы в формате, принятом в Германии, служит приведенный ниже фрагмент кода.

```
Locale loc = new Locale("de", "DE");
NumberFormat currFmt =
      NumberFormat.getCurrencyInstance(loc);
double amt = 123456.78;
String result = currFmt.format(amt);
```

В результате выполнения этого фрагмента кода получается следующая символьная строка:

```
123.456,78 €
```

Для обозначения европейской валюты в данном случае используется знак €, который располагается в конце строки. Обратите также внимание на местоположение знаков, обозначающих дробную часть числа и разделяющих десятичные разряды в его целой части. Порядок их следования обратный принятому в англоязычных странах.

Рассмотрим обратную задачу преобразования в число символьной строки, составленной в соответствии с конкретными региональными настройками. Для этого предусмотрен метод `parse()`, выполняющий синтаксический анализ символьной строки, автоматически используя региональные настройки, заданные по умолчанию. В приведенном ниже примере кода демонстрируется преобразование в число символьной строки, введенной пользователем в текстовом поле. Метод `parse()` способен преобразовывать символьные строки в числа, где в качестве разделителей используются точки и запятые в соответствии с принятыми региональными стандартами.

```
TextField inputField;
. . .

// получить средство форматирования чисел для используемых
// по умолчанию региональных настроек:
NumberFormat fmt = NumberFormat.getNumberInstance();
Number input = fmt.parse(inputField.getText().trim());
double x = input.doubleValue();
```

Метод `parse()` возвращает результат абстрактного типа `Number`. На самом деле возвращаемый объект является экземпляром класса `Long` или `Double` в зависимости от того, представляет ли исходная символьная строка целое число или число с плавающей точкой. Если это не так важно, то для получения упакованного числового значения достаточно вызвать метод `doubleValue()` из класса `Number`.

ВНИМАНИЕ! Для объектов типа **`Number`** не поддерживается автоматическая распаковка, поэтому их нельзя присвоить непосредственно переменным примитивных типов. Вместо этого придется вызвать метод **`doubleValue()`** или **`intValue()`**.

Если число представлено в неверном формате, генерируется исключение типа `ParseException`. Например, не допускается наличие пробелов в начале символьной строки, преобразуемой в число. (Для их удаления следует вызвать метод `trim()`.) Но любые символы, следующие в строке после числа, просто игнорируются, и поэтому исключение в этом случае не возникает.

Следует, однако, иметь в виду, что классы, возвращаемые фабричными методами типа `get`*`Xxx`*`Instance()`, являются экземплярами не абстрактного класса `NumberFormat`, а одного из его подклассов. Фабричным методам известно только, как найти объект, относящийся к конкретным региональным настройкам.

Для получения списка поддерживаемых региональных настроек можно вызвать статический метод `getAvailableLocales()`, возвращающий массив региональных настроек, для которых могут быть получены форматирующие объекты.

В примере программы, рассматриваемой в этом разделе, предоставляется возможность поэкспериментировать с разными способами форматирования чисел (рис. 7.1). В верхней части рабочего окна этой программы находится список всех

доступных региональных настроек со средствами форматирования чисел. Ниже этого списка расположена группа кнопок-переключателей, в которой можно выбрать способ форматирования чисел, денежных сумм или числовых величин в процентах. После выбора новой региональной настройки или способа форматирования чисел число в текстовом поле автоматически переформатируется.

Просмотрев лишь несколько вариантов форматирования чисел и денежных сумм в зависимости от выбранных региональных настроек, пользователь, несомненно, составит ясное представление о разнообразии существующих форматов чисел. В текстовом поле можно ввести любое число и щелкнуть на кнопке Parse, в результате чего будет вызван метод `parse()`, выполняющий синтаксический анализ введенной символьной строки. При удачном исходе синтаксического анализа результат передается методу `format()`, а затем отображается на экране. В противном случае в текстовом поле появляется сообщение `"Parse error"` (Ошибка синтаксического анализа).

Рис. 7.1. Рабочее окно программы **NumberFormatTest**

Как следует из листинга 7.1, исходный код рассматриваемой здесь программы имеет довольно простую структуру. Сначала в конструкторе вызывается метод `NumberFormat.getAvailableLocales()`. Затем для каждого вида поддерживаемых региональных настроек вызывается метод `getDisplayName()`, а возвращаемые этим методом результаты вводятся в список. (Символьные строки не сортируются; подробнее об этом речь пойдет в разделе 7.4.) Всякий раз, когда пользователь выбирает другие региональные настройки или способ форматирования чисел, создается новый форматирующий объект и обновляется содержимое текстового поля. Если пользователь щелкнет на кнопке Parse, то вызывается метод `parse()`, преобразующий в число символьную строку в соответствии с выбранными региональными настройками.

НА ЗАМЕТКУ! Для чтения локализованных целых чисел, а также чисел с плавающей точкой можно воспользоваться классом **Scanner**, вызвав метод **useLocale()** из этого класса для установки региональных настроек.

Листинг 7.1. Исходный код из файла **numberFormat/NumberFormatTest.java**

```
1  package numberFormat;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.text.*;
6  import java.util.*;
```

```
7
8   import javax.swing.*;
9
10  /**
11   * В этой программе демонстрируется форматирование чисел
12   * при разных региональных настройках
13   * @version 1.15 2018-05-01
14   * @author Cay Horstmann
15   */
16  public class NumberFormatTest
17  {
18    public static void main(String[] args)
19    {
20      EventQueue.invokeLater(() ->
21        {
22          var frame = new NumberFormatFrame();
23          frame.setTitle("NumberFormatTest");
24          frame.setDefaultCloseOperation(
25                JFrame.EXIT_ON_CLOSE);
26          frame.setVisible(true);
27        });
28    }
29   }
30
31  /**
32   * Этот фрейм содержит кнопки-переключатели для
33   * выбора способа форматирования чисел, комбинированный
34   * список для выбора региональных настроек, текстовое
35   * поле для отображения отформатированного числа,
36   * а также кнопку для активизации синтаксического
37   * анализа содержимого текстового поля
38   */
39  class NumberFormatFrame extends JFrame
40  {
41    private Locale[] locales;
42    private double currentNumber;
43    private JComboBox<String> localeCombo =
44            new JComboBox<>();
45    private JButton parseButton =
46            new JButton("Parse");
47    private JTextField numberText =
48            new JTextField(30);
49    private JRadioButton numberRadioButton =
50            new JRadioButton("Number");
51    private JRadioButton currencyRadioButton =
52            new JRadioButton("Currency");
53    private JRadioButton percentRadioButton =
54            new JRadioButton("Percent");
55    private ButtonGroup rbGroup = new ButtonGroup();
56    private NumberFormat currentNumberFormat;
57
58    public NumberFormatFrame()
59    {
60      setLayout(new GridBagLayout());
61
62      ActionListener listener = event -> updateDisplay();
63
```

```
64      var p = new JPanel();
65      addRadioButton(p, numberRadioButton,
66                     rbGroup, listener);
67      addRadioButton(p, currencyRadioButton,
68                     rbGroup, listener);
69      addRadioButton(p, percentRadioButton,
70                     rbGroup, listener);
71
72      add(new JLabel("Locale:"),
73          new GBC(0, 0).setAnchor(GBC.EAST));
74      add(p, new GBC(1, 1));
75      add(parseButton, new GBC(0, 2).setInsets(2));
76      add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
77      add(numberText,
78          new GBC(1, 2).setFill(GBC.HORIZONTAL));
79      locales = (Locale[])
80              NumberFormat.getAvailableLocales().clone();
81      Arrays.sort(locales,
82           Comparator.comparing(Locale::getDisplayName));
83      for (Locale loc : locales)
84        localeCombo.addItem(loc.getDisplayName());
85      localeCombo.setSelectedItem(
86                  Locale.getDefault().getDisplayName());
87      currentNumber = 123456.78;
88      updateDisplay();
89
90      localeCombo.addActionListener(listener);
91
92      parseButton.addActionListener(event ->
93        {
94          String s = numberText.getText().trim();
95          try
96          {
97            Number n = currentNumberFormat.parse(s);
98            currentNumber = n.doubleValue();
99            updateDisplay();
100         }
101         catch (ParseException e)
102         {
103           numberText.setText(e.getMessage());
104         }
105       });
106     pack();
107   }
108
109   /**
110    * Вводит кнопки-переключатели в контейнер
111    * @param p Контейнер для размещения
112    *          кнопок-переключателей
113    * @param b Кнопка-переключатель
114    * @param g Группа кнопок-переключателей
115    * @param listener Приемник событий от
116    *                 кнопок-переключателей
117    */
118   public void addRadioButton(Container p,
119              JRadioButton b, ButtonGroup g,
120              ActionListener listener)
```

```
121   {
122     b.setSelected(g.getButtonCount() == 0);
123     b.addActionListener(listener);
124     g.add(b);
125     p.add(b);
126   }
127
128   /**
129    * Обновляет отображаемое число и форматирует его
130    * в соответствии с пользовательскими установками
131    */
132   public void updateDisplay()
133   {
134     Locale currentLocale =
135           locales[localeCombo.getSelectedIndex()];
136     currentNumberFormat = null;
137     if (numberRadioButton.isSelected())
138       currentNumberFormat = NumberFormat
139                   .getNumberInstance(currentLocale);
140     else if (currencyRadioButton.isSelected())
141       currentNumberFormat = NumberFormat
142                   .getCurrencyInstance(currentLocale);
143     else if (percentRadioButton.isSelected())
144       currentNumberFormat = NumberFormat
145                   .getPercentInstance(currentLocale);
146     String formatted = currentNumberFormat
147                        .format(currentNumber);
148     numberText.setText(formatted);
149   }
150 }
```

java.text.NumberFormat 1.1

- **static Locale[] getAvailableLocales()**

  Возвращает массив объектов типа **Locale**, для которых доступны форматирующие объекты типа **NumberFormat**.

- **static NumberFormat getNumberInstance()**
- **static NumberFormat getNumberInstance(Locale *l*)**
- **static NumberFormat getCurrencyInstance()**
- **static NumberFormat getCurrencyInstance(Locale *l*)**
- **static NumberFormat getPercentInstance()**
- **static NumberFormat getPercentInstance(Locale *l*)**

  Возвращают объект, форматирующий числа, денежные суммы или числовые величины в процентах в соответствии с текущими или заданными региональными настройками.

- **String format(double *x*)**
- **String format(long *x*)**

  Возвращают символьную строку, получаемую в результате форматирования заданного числа с плавающей точкой или целого числа.

java.text.NumberFormat 1.1 *(окончание)*

- **Number parse(String *s*)**

  Возвращает число, получаемое в результате синтаксического анализа символьной строки. Это число может иметь тип **Long** или **Double**, а символьная строка не должна начинаться с пробелов. Любые символы, следующие в строке после анализируемого числа, игнорируются. При неудачном исходе синтаксического анализа символьной строки генерируется исключение типа **ParseException**.

- **void setParseIntegerOnly(boolean *b*)**
- **boolean isParseIntegerOnly()**

  Устанавливают или получают признак, указывающий на то, что данный форматирующий объект предназначен для синтаксического анализа только целочисленных значений.

- **void setGroupingUsed(boolean *b*)**
- **boolean isGroupingUsed()**

  Устанавливают или получают признак, указывающий на то, что данный форматирующий объект предназначен для распознавания и разделения групп десятичных разрядов (например, **100,000**) в анализируемых числах.

- **void setMinimumIntegerDigits(int *n*)**
- **int getMinimumIntegerDigits()**
- **void setMaximumIntegerDigits(int *n*)**
- **int getMaximumIntegerDigits()**
- **void setMinimumFractionDigits(int *n*)**
- **int getMinimumFractionDigits()**
- **void setMaximumFractionDigits(int *n*)**
- **int getMaximumFractionDigits()**

  Устанавливают или получают максимальное или минимальное количество цифр в целой или дробной части числа.

7.2.2. Форматирование денежных сумм в разных валютах

Для форматирования денежных сумм служит метод `NumberFormat.getCurrencyInstance()`, но он не очень удобен, поскольку возвращает форматирующий объект только для одной валюты. Допустим, для американского заказчика выписывается счет-фактура, где одни суммы представлены в долларах США, а другие в евро. Для решения этой задачи нельзя просто воспользоваться двумя форматирующими объектами, как показано ниже. Счет, в котором фигурируют такие суммы, как `$100,000` и `100.000€`, будет выглядеть не совсем обычно. Ведь при представлении сумм в евро для разделения групп разрядов используется точка, а сумм в долларах США — запятая.

```
NumberFormat dollarFormatter =
      NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat euroFormatter =
      NumberFormat.getCurrencyInstance(Locale.GERMANY);
```

Для управления форматированием денежных сумм в разных валютах лучше воспользоваться классом `Currency`. Сначала получается объект типа `Currency`,

для чего статическому методу `Currency.getInstance()` передается идентификатор валюты. Затем вызывается метод `setCurrency()` для каждого форматирующего объекта. В приведенном ниже фрагменте кода показано, как подстроить под американского заказчика объект, форматирующий денежные суммы в евро.

```
NumberFormat euroFormatter =
      NumberFormat.getCurrencyInstance(Locale.US);
euroFormatter.setCurrency(Currency.getInstance("EUR"));
```

Идентификаторы валют определяются по стандарту ISO 4217 (`https://www.iso.org/iso-4217-currency-codes.html`). Некоторые из них приведены в табл. 7.3.

Таблица 7.3. Идентификаторы валют

Валюта	Идентификатор
Доллар США	**USD**
Евро	**EUR**
Английский фунт	**GBP**
Японская иена	**JPY**
Китайский юань	**CNY**
Индийская рупия	**INR**
Российский рубль	**RUB**

`java.util.Currency` 1.4

- **`static Currency getInstance(String currencyCode)`**
- **`static Currency getInstance(Locale locale)`**

  Возвращают экземпляр класса **Currency**, соответствующий заданному коду валюты по стандарту ISO 4217 или стране, указанной в текущих региональных настройках.
- **`String toString()`**
- **`String getCurrencyCode()`**
- **`String getNumericCode()`** 7
- **`getNumericCodeAsString()`** 9

  Получают код текущей валюты по стандарту ISO 4217.
- **`String getSymbol()`**
- **`String getSymbol(Locale locale)`**

  Получают форматирующий знак текущей валюты в соответствии с текущими или заданными региональными настройками. Например, доллар США (**USD**) может обозначаться как **$** или **US$** в зависимости от используемых региональных настроек.
- **`int getDefaultFractionDigits()`**

  Получает принятое по умолчанию количество цифр в дробной части денежной суммы, указанной в текущей валюте.
- **`static Set<Currency> getAvailableCurrencies()`** 7

  Получает все имеющиеся валюты.

7.3. Форматирование даты и времени

При форматировании даты и времени в соответствии с региональными настройками необходимо иметь в виду четыре особенности.

- Названия месяцев и дней недели должны быть представлены на местном языке.
- Порядок указания года, месяца и числа отличается в разных странах и регионах.
- Для отображения дат может использоваться календарь, отличный от григорианского.
- Следует учитывать часовые пояса.

Для форматирования даты и времени применяется класс `DateTimeFormatter`. Затем выбирается один из четырех стилей форматирования, перечисленных в табл. 7.4. Далее получается средство форматирования следующим образом:

```
// Один из стилей форматирования FormatStyle.SHORT,
// FormatStyle.MEDIUM, ...
FormatStyle style = ...;
DateTimeFormatter dateFormatter =
        DateTimeFormatter.ofLocalizedDate(style);
DateTimeFormatter timeFormatter =
        DateTimeFormatter.ofLocalizedTime(style);
DateTimeFormatter dateTimeFormatter =
        DateTimeFormatter.ofLocalizedDateTime(style);
        // или DateTimeFormatter
        //      .ofLocalizedDateTime(style1, style2)
```

Таблица 7.4. Стили форматирования даты и времени с учетом региональных настроек

Стиль	Дата	Время
SHORT	`7/16/69`	`9:32 AM`
MEDIUM	`Jul 16, 1969`	`9:32:00 AM`
LONG	`July 16, 1969`	`9:32:00 AM EDT` с учетом региональных настроек **en-US**, `9:32:00 MSZ` с учетом региональных настроек **de-DE** (только для класса **ZonedDateTime**)
FULL	`Wednesday, July 16, 1969`	`9:32:00 AM EDT` с учетом региональных настроек **en-US**, `9:32 Uhr MSZ` с учетом региональных настроек **de-DE** (только для класса **ZonedDateTime**)

В этих средствах форматирования используются текущие региональные настройки форматов даты и времени. Чтобы выбрать другие региональные настройки, достаточно вызвать метод `withLocale()` следующим образом:

```
DateTimeFormatter dateFormatter = DateTimeFormatter
            .ofLocalizedDate(style).withLocale(locale);
```

Теперь можно отформатировать местную дату (объект типа `LocalDate`), местное время и дату (объект типа `LocalDateTime`), местное время (объект типа `LocalTime`) или поясное время и дату (объект типа `ZonedDateTime`), как показано ниже.

```
ZonedDateTime appointment = ...;
String formatted = formatter.format(appointment);
```

НА ЗАМЕТКУ! В данном случае применяется класс **DateTimeFormatter** из пакета **java.time**. Имеется также устаревший класс **java.text.DateFormatter**, внедренный еще в версии Java 1.1 для манипулирования объектами типа **Date** и **Calendar**.

Для синтаксического анализа символьной строки, содержащей дату и время, служит один из статических методов `parse()` в классах `LocalDate`, `LocalDateTime`, `LocalTime` или `ZonedDateTime`:

```
LocalTime time = LocalTime.parse("9:32 AM", formatter);
```

Но методы `parse()` из упомянутых выше классов непригодны для синтаксического анализа данных, вводимых пользователем, по крайней мере, для их предварительной обработки. Например, средство форматирования даты и времени в кратком стиле для Соединенных Штатов способно проанализировать символьную строку `"9:32 AM"`, но не строку `"9:32AM"` или `"9:32 am"`.

ВНИМАНИЕ! Средства форматирования дат подвергают синтаксическому анализу несуществующие даты вроде 31 ноября, корректируя их по последней дате в данном месяце.

Иногда в календарном приложении требуется отображать, например, только наименования дней недели и месяцы. С этой целью можно вызвать метод `getDisplayName()` из перечислений `DayOfWeek` и `Month`, как показано ниже.

```
for (Month m : Month.values())
  System.out.println(m.getDisplayName(textStyle, locale) + " ");
```

Стили форматирования текста перечислены в табл. 7.5. Стили типа `STANDALONE` служат для отображения за пределами форматируемой даты. Например, январь по-фински обозначается как "tammikuuta" в самой дате, но как "tammikuu" за ее пределами или отдельно.

Таблица 7.5. Стили форматирования текста, представленные константами из перечисления `java.time.format.TextStyle`

Стиль	Пример
FULL / FULL_STANDALONE	**January**
SHORT / SHORT_STANDALONE	**Jan**
NARROW / NARROW STANDALONE	**J**

НА ЗАМЕТКУ! Первым днем недели может быть суббота, воскресенье или понедельник в зависимости от конкретных региональных настроек. Выяснить первый день недели с учетом региональных настроек можно следующим образом:

```
DayOfWeek first = WeekFields.of(locale).getFirstDayOfWeek();
```

В примере программы, исходный код которой приведен в листинге 7.2, демонстрируется применение класса `DateFormat` на практике. Эта программа позволяет выбирать различные региональные настройки и наблюдать за тем, как в разных странах форматируются дата и время. На рис. 7.2 показано рабочее окно данной программы после установки китайских шрифтов на компьютере. Как видите, даты выводятся на экран в правильном формате для китайских региональных настроек.

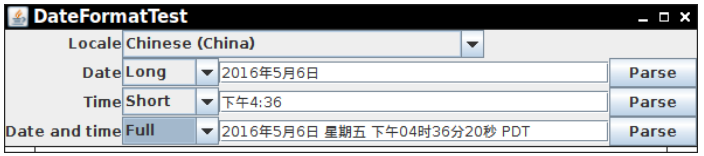


Рис. 7.2. Отображение даты на китайском языке в рабочем окне программы **DateFormatTest**

Листинг 7.2. Исходный код из файла **dateFormat/DateFormatTest.java**

```
1  package dateFormat;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.time.*;
6  import java.time.format.*;
7  import java.util.*;
8
9  import javax.swing.*;
10
11 /**
12  * В этой программе демонстрируется форматирование дат
13  * при выборе разных региональных настроек
14  * @version 1.01 2018-05-01
15  * @author Cay Horstmann
16  */
17 public class DateTimeFormatterTest
18 {
19   public static void main(String[] args)
20   {
21     EventQueue.invokeLater(() ->
22       {
23         var frame = new DateTimeFormatterFrame();
24         frame.setTitle("DateFormatTest");
25         frame.setDefaultCloseOperation(
26               JFrame.EXIT_ON_CLOSE);
27         frame.setVisible(true);
28       });
29   }
30 }
31
32 /**
33  * Этот фрейм содержит комбинированные списки для
```

```
34   * выбора региональных настроек и форматов даты и
35   * времени, текстовые поля для отображения
36   * отформатированных даты и времени, а также кнопки
37   * синтаксического анализа содержимого текстовых
38   * полей и флажок для установки режима нестрогой
39   * интерпретации вводимых дат и времени
40   */
41  class DateTimeFormatterFrame extends JFrame
42  {
43    private Locale[] locales;
44    private LocalDate currentDate;
45    private LocalTime currentTime;
46    private ZonedDateTime currentDateTime;
47    private DateTimeFormatter currentDateFormat;
48    private DateTimeFormatter currentTimeFormat;
49    private DateTimeFormatter currentDateTimeFormat;
50    private JComboBox<String> localeCombo =
51            new JComboBox<>();
52    private JButton dateParseButton =
53            new JButton("Parse");
54    private JButton timeParseButton =
55            new JButton("Parse");
56    private JButton dateTimeParseButton =
57            new JButton("Parse");
58    private JTextField dateText = new JTextField(30);
59    private JTextField timeText = new JTextField(30);
60    private JTextField dateTimeText = new JTextField(30);
61    private EnumCombo<FormatStyle> dateStyleCombo =
62            new EnumCombo<>(FormatStyle.class, "Short",
63                            "Medium", "Long", "Full");
64    private EnumCombo<FormatStyle> timeStyleCombo =
65            new EnumCombo<>(FormatStyle.class,
66                            "Short", "Medium");
67    private EnumCombo<FormatStyle> dateTimeStyleCombo =
68            new EnumCombo<>(FormatStyle.class, "Short",
69                            "Medium", "Long", "Full");
70
71    public DateTimeFormatterFrame()
72    {
73      setLayout(new GridBagLayout());
74      add(new JLabel("Locale"),
75          new GBC(0, 0).setAnchor(GBC.EAST));
76      add(localeCombo, new GBC(1, 0, 2, 1)
77                          .setAnchor(GBC.WEST));
78
79      add(new JLabel("Date"),
80          new GBC(0, 1).setAnchor(GBC.EAST));
81      add(dateStyleCombo,
82          new GBC(1, 1).setAnchor(GBC.WEST));
83      add(dateText,
84          new GBC(2, 1, 2, 1).setFill(GBC.HORIZONTAL));
85      add(dateParseButton,
86          new GBC(4, 1).setAnchor(GBC.WEST));
87
88      add(new JLabel("Time"),
89          new GBC(0, 2).setAnchor(GBC.EAST));
```

```
90      add(timeStyleCombo,
91          new GBC(1, 2).setAnchor(GBC.WEST));
92      add(timeText,
93          new GBC(2, 2, 2, 1).setFill(GBC.HORIZONTAL));
94      add(timeParseButton,
95          new GBC(4, 2).setAnchor(GBC.WEST));
96
97      add(new JLabel("Date and time"),
98                  new GBC(0, 3).setAnchor(GBC.EAST));
99      add(dateTimeStyleCombo,
100         new GBC(1, 3).setAnchor(GBC.WEST));
101     add(dateTimeText,
102         new GBC(2, 3, 2, 1).setFill(GBC.HORIZONTAL));
103     add(dateTimeParseButton,
104         new GBC(4, 3).setAnchor(GBC.WEST));
105
106     locales = (Locale[])
107               Locale.getAvailableLocales().clone();
108     Arrays.sort(locales, Comparator.comparing(
109                          Locale::getDisplayName));
110     for (Locale loc : locales)
111       localeCombo.addItem(loc.getDisplayName());
112     localeCombo.setSelectedItem(
113                Locale.getDefault().getDisplayName());
114     currentDate = LocalDate.now();
115     currentTime = LocalTime.now();
116     currentDateTime = ZonedDateTime.now();
117     updateDisplay();
118
119     ActionListener listener = event -> updateDisplay();
120     localeCombo.addActionListener(listener);
121     dateStyleCombo.addActionListener(listener);
122     timeStyleCombo.addActionListener(listener);
123     dateTimeStyleCombo.addActionListener(listener);
124
125     addAction(dateParseButton, () ->
126       {
127         currentDate = LocalDate.parse(
128                       dateText.getText().trim(),
129                       currentDateFormat);
130       });
131     addAction(timeParseButton, () ->
132       {
133         currentTime = LocalTime.parse(
134                       timeText.getText().trim(),
135                       currentTimeFormat);
136       });
137     addAction(dateTimeParseButton, () ->
138       {
139         currentDateTime = ZonedDateTime.parse(
140                         dateTimeText.getText().trim(),
142                         currentDateTimeFormat);
143       });
144
145     pack();
146   }
```

```
147
148   /**
149    * Добавляет заданное действие к экранной кнопке, а
150    * по завершении обновляет отображение
151    * @param button Экранная кнопка, к которой
152    *               добавляется действие
153    * @param action Действие, выполняемое при щелчке
154    *               на экранной кнопке
155    */
156   public void addAction(JButton button, Runnable action)
157   {
158     button.addActionListener(event ->
159       {
160         try
161         {
162           action.run();
163           updateDisplay();
164         }
165         catch (Exception e)
166         {
167           JOptionPane.showMessageDialog(
168                       null, e.getMessage());
169         }
170       });
171   }
172
173   /**
174    * Обновляет отображаемые дату и время и форматирует
175    * их в соответствии с пользовательскими установками
176    */
177   public void updateDisplay()
178   {
179     Locale currentLocale =
180            locales[localeCombo.getSelectedIndex()];
181     FormatStyle dateStyle = dateStyleCombo.getValue();
182     currentDateFormat =
183            DateTimeFormatter.ofLocalizedDate(dateStyle)
184                             .withLocale(currentLocale);
185     dateText.setText(currentDateFormat
186                      .format(currentDate));
187     FormatStyle timeStyle = timeStyleCombo.getValue();
188     currentTimeFormat =
189            DateTimeFormatter.ofLocalizedTime(timeStyle)
190                             .withLocale(currentLocale);
191     timeText.setText(
192              currentTimeFormat.format(currentTime));
193     FormatStyle dateTimeStyle =
194                dateTimeStyleCombo.getValue();
195     currentDateTimeFormat =
196           DateTimeFormatter.ofLocalizedDateTime(
197               dateTimeStyle).withLocale(currentLocale);
198     dateTimeText.setText(currentDateTimeFormat
199                          .format(currentDateTime));
200   }
201 }
```

Для проверки правильности синтаксического анализа и преобразования символьной строки в дату достаточно ввести дату, время или и то и другое, а затем щелкнуть на кнопке Parse (Произвести синтаксический анализ). В рассматриваемом здесь примере программы используется вспомогательный класс `EnumCombo`, исходный код которого приведен в листинге 7.3. Он служит для заполнения комбинированного списка значениями типа `Short`, `Medium` и `Long`, а также для автоматического преобразования выбранного пользователем варианта в значение `FormatStyle.SHORT`, `FormatStyle.MEDIUM` или `FormatStyle.LONG`. Чтобы не писать повторяющийся код, в данном случае применяется рефлексия. Выбранный пользователем вариант преобразуется в верхний регистр, пробелы заменяются символами подчеркивания, после чего определяется значение в статическом поле с полученным в итоге именем. (Более подробно рефлексия рассматривается в главе 5 первого тома настоящего издания.)

Листинг 7.3. Исходный код из файла **dateFormat/EnumCombo.java**

```
1  package dateFormat;
2
3  import java.util.*;
4  import javax.swing.*;
5
6  /**
7   * Комбинированный список для выбора среди значений
8   * статических полей, имена которых задаются в
9   * конструкторе вспомогательного класса
10  * @version 1.15 2016-05-06
11  * @author Cay Horstmann
12  */
13 public class EnumCombo<T> extends JComboBox<String>
14 {
15   private Map<String, T> table = new TreeMap<>();
16
17   /**
18    * Конструирует объект вспомогательного
19    * класса EnumCombo, производящий значения типа T
20    * @param cl Класс
21    * @param labels Массив символьных строк, описывающих
22    *               имена статических полей из класса cl,
23    *               относящихся к типу T
24    */
25   public EnumCombo(Class<?> cl, String... labels)
26   {
27     for (String label : labels)
28     {
29       String name = label.toUpperCase()
30                          .replace(' ', '_');
31       try
32       {
33         java.lang.reflect.Field f = cl.getField(name);
34         @SuppressWarnings("unchecked")
35                  T value = (T) f.get(cl);
36         table.put(label, value);
37       }
```

```
38        catch (Exception e)
39        {
40          label = "(" + label + ")";
41          table.put(label, null);
42        }
43        addItem(label);
44      }
45       setSelectedItem(labels[0]);
46    }
47
48    /**
49     * Возвращает значение поля, выбранного пользователем
50     * @return Значение статического поля
51     */
52    public T getValue()
53    {
54      return table.get(getSelectedItem());
55    }
56 }
```

java.time.format.DateTimeFormatter 8

- **static DateTimeFormatter ofLocalizedDate(FormatStyle *dateStyle*)**
- **static DateTimeFormatter ofLocalizedTime(FormatStyle *dateStyle*)**
- **static DateTimeFormatter ofLocalizedDateTime(FormatStyle *dateTimeStyle*)**
- **static DateTimeFormatter ofLocalizedDate(FormatStyle *dateStyle*, FormatStyle *timeStyle*)**

  Возвращают экземпляры типа **DateTimeFormatter** для форматирования дат, времени или того и другого с учетом заданных стилей.
- **DateTimeFormatter withLocale(Locale *locale*)**

  Возвращает копию данного средства форматирования вместе с заданными региональными настройками.
- **String format(TemporalAccessor *temporal*)**

  Возвращает символьную строку, получающуюся в результате форматирования заданных даты и времени.

java.time.LocalDate 8
java.time.LocalTime 8
java.time.LocalDateTime 8
java.time.ZonedDateTime 8

- **static *Xxx* parse(CharSequence *text*, DateTimeFormatter *formatter*)**

  Производит синтаксический анализ заданной символьной строки и возвращает описанную в ней местную дату или время в виде объекта типа **LocalDate**, **LocalTime**, **LocalDateTime** или **ZonedDateTime**. Генерирует исключение типа **DateTimeParseException** при неудачном исходе синтаксического анализа.

7.4. Сортировка и нормализация

Как известно, для сравнения символьных строк служит метод `compareTo()` из класса `String`. К сожалению, этот метод не совсем годится для взаимодействия с пользователями. В методе `compareTo()` применяются строковые значения в кодировке UTF-16, что приводит к абсурдным результатам, даже на английском языке. Например, следующие пять символьных строк упорядочиваются по результатам сортировки методом `compareTo()` таким образом:

```
Athens
Zulu
able
zebra
Engström
```

При упорядочении словаря приходится учитывать регистр букв, но совсем необязательно ударение. Для англоязычного пользователя приведенный выше перечень слов должен быть упорядочен следующим образом:

```
able
Engström
Athens
zebra
Zulu
```

Но такой порядок следования слов неприемлем для шведскоязычного пользователя. Ведь в шведском языке буква `Å` отличается от буквы `A` и поэтому сортируется *после* буквы `Z`! Это означает, что для шведскоязычного пользователя упомянутый выше перечень слов должен быть отсортирован следующим образом:

```
able
Athens
zebra
Zulu
Ångström
```

Чтобы получить компаратор с учетом региональных настроек, следует вызвать метод `Collator.getInstance()`, как показано ниже. Класс `Collator` реализует интерфейс `Comparator`, поэтому объект типа `Collator` можно передать методу `List.sort(Comparator)`, чтобы отсортировать символьные строки.

```
// Класс Collator реализует интерфейс Comparator<Object>:
Collator coll = Collator.getInstance(locale);
words.sort(coll);
```

Для средств сортировки предусмотрены четыре уровня избирательности: *первостепенный*, *второстепенный*, *третьестепенный* и *идентичный*. Например, в английском языке отличие букв `A` и `Z` считается первостепенным, букв `A` и `Å` — второстепенным, а букв `A` и `a` — третьестепенным.

Для того чтобы при сортировке внимание обращалось только на первостепенные отличия, следует задать уровень ее избирательности `Collator.PRIMARY`. Если задать уровень избирательности `Collator.SECONDARY`, то будут учтены и второстепенные отличия. Таким образом, вероятность найти отличия в двух символьных строках будет больше при установке более высокого уровня избирательности, как показано в табл. 7.6.

Таблица 7.6. Сортировка с разными уровнями избирательности (английские региональные настройки)

Первостепенный уровень	Второстепенный уровень	Третьестепенный уровень
Angstrom = Ångström	Angstrom ≠ Ångström	Angstrom ≠ Ångström
Able = able	Able = able	Able ≠ able

Если же установлен уровень избирательности `Collator.IDENTICAL`, то отличия не допускаются. Этот уровень избирательности используется главным образом вместе с *режимом разложения на составляющие*, который устанавливается для сортировки и рассматривается ниже.

Иногда символ или последовательность символов могут быть описаны не только в Юникоде. Например, символу Å в Юникоде соответствует код `U+00C5`. С другой стороны, его можно представить в виде последовательности символов `A` (код **U+0065**) и ° (кружок сверху; код `U+030A`). Еще удивительнее, что последовательность букв "ffi" может быть описана одним символом "латинская малая лигатура ffi" с кодом `U+FB03`. (Можно, конечно, спорить, что это вопрос представления символов, решение которого не должно приводить к появлению разных символов в Юникоде, но правила установлены не нами.)

В стандарте на Юникод определяются четыре *формы нормализации* символьных строк (D, KD, C и KC; подробнее об этом см. по адресу http://www.unicode.org/reports/tr15/tr15-23.html). Две из этих форм используются для сортировки. В форме нормализации D символы с ударением раскладываются на составляющие их буквы и ударения. Например, символ Å раскладывается на составляющие символы `A` и °. А в формах нормализации KC и KD на составляющие раскладываются такие символы, как лигатура ffi или знак торговой марки ™.

Для сортировки можно выбрать определенную степень нормализации. Так, если установить значение константы `Collator.NO_DECOMPOSITION`, то символьные строки вообще не будут нормализованы при сортировке. В этом режиме сортировка выполняется быстрее, но он может быть непригодным для сортировки текста, где символы выражаются во многих формах. По умолчанию устанавливается значение константы `Collator.CANONICAL_DECOMPOSITION`, определяющее режим, в котором используется форма нормализации D. Это самая полезная форма для сортировки текста, содержащего символы с ударениями, но не лигатуры. Наконец, в режиме полного разложения на составляющие используется форма нормализации KD. Характерные примеры сортировки в режимах разложения на составляющие приведены в табл. 7.7.

Таблица 7.7. Сортировка в разных режимах разложения на составляющие

Без разложения на составляющие	Каноническое разложение на составляющие	Полное разложение на составляющие
Å ≠ A°	Å = A°	Å = A
™ ≠ TM	™ ≠ TM	™ = TM

Если одна символьная строка сравнивается многократно с другими строками, то во избежание ее повторного разложения на составляющие и ради повышения эффективности результат разложения на составляющие следует сохранить в объекте *ключа сортировки*. Например, в приведенном ниже фрагменте кода метод `getCollationKey()` возвращает объект типа `CollationKey`, используемый для ускорения всех последующих операций сравнения.

```
String a = . . .;
CollationKey aKey = coll.getCollationKey(a);
// быстрое сравнение:
if(aKey.compareTo(coll.getCollationKey(b)) == 0)
  . . .
```

Наконец, символьные строки иногда требуется преобразовать в их нормализованные формы, не прибегая к сортировке. Такая потребность возникает, например, при сохранении символьных строк в базе данных или при взаимодействии с другой программой. Для этой цели служит класс `java.text.Normalizer`, выполняющий процесс нормализации, как показано ниже. Нормализованная строка содержит десять символов. Символы Å и ö заменяются последовательностями символов "A°" и "ö".

```
String name = "Ångström";
// использовать форму нормализации D:
String normalized = Normalizer.normalize(
                    name, Normalizer.Form.NFD);
```

Тем не менее это обычно не самая лучшая форма для хранения и передачи символьных строк. В форме нормализации C сначала выполняется разложение на составляющие, а затем в установленном порядке присоединяются ударения. В соответствии с рекомендациями консорциума W3C такой режим является наиболее предпочтительным для передачи данных через Интернет.

Программа, исходный код которой приведен в листинге 7.4, позволяет экспериментировать с разными видами сортировки. Достаточно ввести слово в текстовом поле и щелкнуть на кнопке Add, чтобы добавить введенное слово в список. Список сортируется заново после добавления в него каждого слова, изменения региональных настроек (в раскрывающемся списке Locale), уровня избирательности сортировки (в раскрывающемся списке Strength) или режима разложения на составляющие (в раскрывающемся списке Decomposition). Знак равенства (=) обозначает, что слова считаются одинаковыми (рис. 7.3).

Наименования региональных настроек в раскрывающемся списке Locale отображаются в порядке, отсортированном в соответствии с устанавливаемыми по умолчанию региональными настройками. Так, если запустить рассматриваемую здесь программу при стандартных региональных настройках `US English`, то региональные настройки `Norwegian (Norway, Nynorsk)` окажутся выше в данном списке, чем региональные настройки `Norwegian (Norway)`, несмотря на то, что значение знака запятой в Юникоде больше, чем значение знака закрывающей скобки.

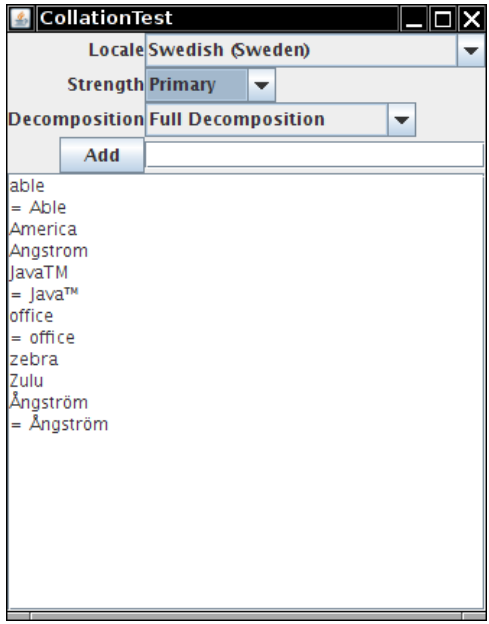


Рис. 7.3. Рабочее окно программы `CollationTest`

Листинг 7.4. Исходный код из файла `collation/CollationTest.java`

```
1   package collation;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.text.*;
6   import java.util.*;
7   import java.util.List;
8
9   import javax.swing.*;
10
11  /**
12   * В этой программе демонстрируется сортировка
13   * символьных строк при выборе разных региональных
14   * настроек
15   * @version 1.16 2018-05-01
16   * @author Cay Horstmann
17   */
18  public class CollationTest
19  {
20    public static void main(String[] args)
21    {
22      EventQueue.invokeLater(() ->
23        {
24          var frame = new CollationFrame();
25          frame.setTitle("CollationTest");
26          frame.setDefaultCloseOperation(
```

```
27                JFrame.EXIT_ON_CLOSE);
28          frame.setVisible(true);
29        });
30    }
31  }
32
33  /**
34   * Этот фрейм содержит комбинированные списки для
35   * выбора региональных настроек, уровня избирательности
36   * сортировки и режимов разложения на составляющие,
37   * текстовое поле и кнопку для ввода новых символьных
38   * строк, а также текстовую область для перечисления
39   * отсортированных символьных строк
40   */
41  class CollationFrame extends JFrame
42  {
43    private Collator collator = Collator.getInstance(
44                                Locale.getDefault());
45    private List<String> strings = new ArrayList<>();
46    private Collator currentCollator;
47    private Locale[] locales;
48    private JComboBox<String> localeCombo =
49                              new JComboBox<>();
50    private JTextField newWord = new JTextField(20);
51    private JTextArea sortedWords = new JTextArea(20, 20);
52    private JButton addButton = new JButton("Add");
53    private EnumCombo<Integer> strengthCombo =
54            new EnumCombo<>(Collator.class, "Primary",
55                            "Secondary", "Tertiary",
56                            "Identical");
57    private EnumCombo<Integer> decompositionCombo =
58            new EnumCombo<>(Collator.class,
59                            "Canonical Decomposition",
60                            "Full Decomposition",
61                            "No Decomposition");
62
63    public CollationFrame()
64    {
65      setLayout(new GridBagLayout());
66      add(new JLabel("Locale"),
67          new GBC(0, 0).setAnchor(GBC.EAST));
68      add(new JLabel("Strength"),
69          new GBC(0, 1).setAnchor(GBC.EAST));
70      add(new JLabel("Decomposition"),
71                    new GBC(0, 2).setAnchor(GBC.EAST));
72      add(addButton, new GBC(0, 3).setAnchor(GBC.EAST));
73      add(localeCombo,
74          new GBC(1, 0).setAnchor(GBC.WEST));
75      add(strengthCombo,
76          new GBC(1, 1).setAnchor(GBC.WEST));
77      add(decompositionCombo,
78          new GBC(1, 2).setAnchor(GBC.WEST));
79      add(newWord, new GBC(1, 3).setFill(GBC.HORIZONTAL));
80      add(new JScrollPane(sortedWords),
81          new GBC(0, 4, 2, 1).setFill(GBC.BOTH));
82
```

```
83      locales = (Locale[]) Collator.getAvailableLocales()
84                                  .clone();
85      Arrays.sort(locales, (l1, l2) ->
86                  collator.compare(l1.getDisplayName(),
87                  l2.getDisplayName()));
88      for (Locale loc : locales)
89        localeCombo.addItem(loc.getDisplayName());
90      localeCombo.setSelectedItem(
91                  Locale.getDefault().getDisplayName());
92
93      strings.add("America");
94      strings.add("able");
95      strings.add("Zulu");
96      strings.add("zebra");
97      strings.add("\u00C5ngstr\u00F6m");
98      strings.add("A\u030angstro\u0308m");
99      strings.add("Angstrom");
100     strings.add("Able");
101     strings.add("office");
102     strings.add("o\uFB03ce");
103     strings.add("Java\u2122");
104     strings.add("JavaTM");
105     updateDisplay();
106
107     addButton.addActionListener(event ->
108       {
109         strings.add(newWord.getText());
110         updateDisplay();
111       });
112
113     ActionListener listener = event -> updateDisplay();
114
115     localeCombo.addActionListener(listener);
116     strengthCombo.addActionListener(listener);
117     decompositionCombo.addActionListener(listener);
118     pack();
119   }
120
121   /**
122    * Обновляет отображаемые строки и сортирует их
123    * в соответствии с пользовательскими установками
124    */
125   public void updateDisplay()
126   {
127     Locale currentLocale =
128           locales[localeCombo.getSelectedIndex()];
129     localeCombo.setLocale(currentLocale);
130
131     currentCollator =
132           Collator.getInstance(currentLocale);
133     currentCollator.setStrength(
134                   strengthCombo.getValue());
135     currentCollator.setDecomposition(
136     decompositionCombo.getValue());
137
138     strings.sort(currentCollator);
```

```
139
140     sortedWords.setText("");
141     for (int i = 0; i < strings.size(); i++)
142     {
143       String s = strings.get(i);
144       if (i > 0 && currentCollator.compare(s,
145                   strings.get(i - 1)) == 0)
146         sortedWords.append("= ");
147       sortedWords.append(s + "\n");
148     }
149     pack();
150   }
151 }
```

java.text.Collator 1.1

- **static Locale[] getAvailableLocales()**

  Возвращает массив объектов типа **Locale**, для которых существуют сортирующие объекты типа **Collator**.

- **static Collator getInstance()**
- **static Collator getInstance(Locale *l*)**

  Возвращают объект типа **Collator** для текущих или заданных региональных настроек.

- **int compare(String *a*, String *b*)**

  Возвращает отрицательное значение, если строка ***a*** предшествует строке ***b***; нулевое значение, если строки считаются одинаковыми; или положительное значение, если строка ***b*** предшествует строке ***a***.

- **boolean equals(String *a*, String *b*)**

  Возвращает логическое значение **true**, если строки ***a*** и ***b*** считаются одинаковыми, а иначе — логическое значение **false**.

- **void setStrength(int *strength*)**
- **int getStrength()**

  Устанавливают или получают уровень избирательности сортировки. Чем выше уровень избирательности, тем больше вероятность того, что сортируемые слова будут признаны разными. Поддерживаются следующие уровни избирательности сортировки: **Collator.PRIMARY**, **Collator.SECONDARY** и **Collator.TERTIARY**.

- **void setDecomposition(int *decomp*)**
- **int getDecompositon()**

  Устанавливают или получают режим разложения на составляющие при сортировке символьных строк. Чем выше степень разложения на составляющие, тем строже выполняется сравнение сортируемых символьных строк. Поддерживаются следующие режимы разложения на составляющие: **Collator.NO_DECOMPOSITION**, **Collator.CANONICAL_DECOMPOSITION** и **Collator.FULL_DECOMPOSITION**.

- **CollationKey getCollationKey(String *a*)**

  Возвращает ключ сортировки с разложенными на составляющие символами, чтобы быстро сравнить их по другому ключу сортировки.

java.text.CollationKey 1.1

- **int compareTo(CollationKey *b*)**
  Возвращает отрицательное значение, если данный ключ сортировки предшествует ключу ***b***; нулевое значение, если ключи одинаковы; или положительное значение, если данный ключ следует за ключом ***b***.

java.text.Normalizer 6

- **static String normalize(CharSequence *str*, Normalizer.Form *form*)**
  Возвращает нормализованную форму символьной строки ***str***. Параметр ***form*** может принимать одно из следующих значений: **ND**, **NKD**, **NC** или **NKC**.

7.5. Форматирование сообщений

В состав библиотеки Java входит класс `MessageFormat` для форматирования текста, содержащего фрагменты с переменными. Этот механизм подобен форматированию с помощью метода `printf()`, но он действует с учетом региональных настроек, а также форматов чисел и дат. В последующих разделах этот механизм рассматривается более подробно.

7.5.1. Форматирование чисел и дат

Ниже приведен пример типичной строки форматирования сообщений, где номера в фигурных скобках служат в качестве заполнителей для подлинных имен и значений.

```
"On {2}, a {0} destroyed {1} houses and
 caused {3} of damage."
```

Подставить значения переменных можно с помощью статического метода `MessageFormat.format()` с переменным числом параметров, где подстановка значений переменных может быть произведена следующим образом:

```
String msg = MessageFormat.format(
      "On {2}, a {0} destroyed {1} houses and
          caused {3} of damage.", "hurricane", 99,
      new GregorianCalendar(1999, 0, 1).getTime(), 10.0E8);
```

В данном примере заполнитель `{0}` замещается строковым значением `"hurricane"`, заполнитель `{1}` — числовым значением `99` и т.д. А в результате подстановки получается следующая текстовая строка:

```
On 1/1/99 12:00 AM, a hurricane destroyed 99 houses and
   caused 100,000,000 of damage.
```

Результат для начала неплохой, но вряд ли может устроить полностью. В частности, время `12:00 AM` отображать не следует, а сумму ущерба от урагана нужно представить в денежных единицах. Это можно сделать, указав формат для некоторых переменных, как выделено ниже полужирным.

```
"On {2,date,long}, a {0} destroyed {1} houses and
    caused {3,number,currency} of damage."
```

В результате очередной подстановки получается следующая строка:

```
On January 1, 1999, a hurricane destroyed 99 houses and
   caused $100,000,000 of damage.
```

Обычно после заполнителя допускается задавать *тип* и *стиль*, разделяя их запятыми. Ниже перечислены допустимые типы.

```
number
time
date
choice
```

Если указан тип `number`, то допускаются следующие стили:

```
integer
currency
percent
```

В качестве стиля может быть также указан шаблон числового формата, например `$,##0`. (Подробнее об этом см. в документации на класс `DecimalFormat`.)

Для типа `time` или `date` может быть указан один из следующих стилей:

```
short
medium
long
full
```

Аналогично числам, в качестве стиля может быть указан шаблон даты, например `гггг-мм-дд`. (Допустимые форматы подробно рассматриваются в документации на класс `SimpleDateFormat`.)

ВНИМАНИЕ! Статический метод **`format()`** форматирует значения с учетом текущих региональных настроек. Форматировать сообщения средствами класса **`MessageFormat`** с учетом произвольных региональных настроек немного сложнее, поскольку в этом классе отсутствует метод с переменным числом аргументов. Поэтому форматируемые значения придется разместить в массиве **`Object[]`**, как показано ниже.

```
var mf = new MessageFormat(pattern, loc);
String msg = mf.format(new Object[] { значения });
```

`java.text.MessageFormat` 1.1

- **`Locale getLocale()`**

  Устанавливают или получают региональные настройки для заполнителей в сообщении. Региональные настройки пригодны *только* для последующих шаблонов, задаваемых с помощью метода **`applyPattern()`**.

- **`static String format(String pattern, Object... args)`**

  Форматирует символьную строку по шаблону ***`pattern`***, подставляя вместо заполнителей **`{i}`** объекты из массива **`args[i]`**.

java.text.MessageFormat 1.1 *(окончание)*

- **StringBuffer format(Object *args*, StringBuffer *result*, FieldPosition *pos*)**

  Форматирует шаблон данного объекта типа **MessageFormat**. Параметр ***args*** принимает массив объектов. Форматируемая строка добавляется к значению параметра ***result***, которое затем возвращается. Если параметр ***pos*** принимает ссылку на новый объект **new FieldPosition(MessageFormat.Field.ARGUMENT)**, его свойства **beginIndex** и **endIndex** устанавливаются в соответствии с расположением текста, который подставляется вместо заполнителя **{1}**. Если же сведения о расположении подстановочного текста не важны, в качестве параметра ***pos*** следует задать пустое значение **null**.

java.text.Format 1.1

- **String format(Object *obj*)**

  Форматирует заданный объект по правилам, определяемым текущим форматирующим объектом. С этой целью делается следующий вызов: **format(obj, new StringBuffer(), new Field Position(1)).toString()**.

7.5.2. Форматы выбора

Вернемся к шаблону из предыдущего раздела, чтобы рассмотреть его подробнее:

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage."
```

Если вместо заполнителя `{0}`, обозначающего вид стихийного бедствия, подставить строковое значение `"earthquake"` (землетрясение), то получится следующее предложение, нарушающее правила английской грамматики в отношении используемых артиклей:

```
On January 1, 1999, a earthquake destroyed . . .
```

Для устранения этой грамматической ошибки артикль `a` придется ввести в заполнитель `{0}` следующим образом:

```
"On {2}, {0} destroyed {1} houses and caused {3} of damage."
```

Теперь вместо заполнителя `{0}` будет подставлен текст `"a hurricane"` или `"an earthquake"`. Такой способ особенно удобен для перевода сообщений на языки, в которых в каждом роде употребляется отдельный артикль. Например, на немецком языке этот шаблон должен выглядеть так:

```
"{0} zerstörte am {2} {1} Häuser und richtete einen Schaden
    von {3} an."
```

В этом случае заполнитель будет заменяться грамматически правильными сочетаниями артикля и имени существительного, например `"Ein Wirbelsturm"` и `"Eine Naturkatastrophe"`.

Теперь рассмотрим заполнитель `{1}`. Если стихийное бедствие оказалось не очень разрушительным, то вместо этого заполнителя можно подставить значение `1`. Но и в этом случае получится предложение с нарушением правил английской грамматики:

```
On January 1, 1999, a mudslide destroyed 1 houses and . . .
```

Желательно, чтобы текст сообщения грамотно изменялся в соответствии с одним из следующих подставляемых значений:

```
no houses
one house
2 houses
. . .
```

Именно для этой цели и был внедрен формат выбора типа `choice`. В соответствии с этим форматом задается последовательность пар значений, каждая из которых содержит *нижний предел* и *форматирующую строку*. Нижний предел и форматирующая строка разделяются знаком #, а для разделения пар значений служит знак |. Ниже приведен пример заполнителя `{1}`, в котором формат выбора выделен полужирным. Результаты форматирования текста сообщения в зависимости от значения, подставляемого вместо заполнителя `{1}`, представлены в табл. 7.8.

```
{1, choice, 0#no houses|1#one house|2#{1} houses}
```

Таблица 7.8. Текст сообщения, отформатированный по выбору

{1}	Результат
0	"no houses"
1	"one house"
3	"3 houses"
-1	"no houses"

А зачем в форматирующей строке дважды указывается заполнитель `{1}`? Когда к этому заполнителю применяется формат выбора и значение оказывается равным 2, возвращается символьная строка `"{1} houses"`. Эта строка форматируется еще раз и включается в результирующую строку сообщения.

НА ЗАМЕТКУ! Приведенный выше пример показывает, что разработчики формата выбора приняли не самое лучшее решение. Так, если имеются три форматирующие строки, то для их разделения требуются два предела. Как правило, количество пределов должно быть на *единицу меньше*, чем количество форматирующих строк. И как следует из табл. 7.8, первый предел в классе **`MessageFormat`** вообще игнорируется. Синтаксис форматирования сообщений мог бы быть более понятным, если бы пределы указывались *между* выбираемыми вариантами, например, следующим образом:

```
no houses|1|one house|2|{1} houses
    // более понятный, но не действующий формат
```

С помощью знака < можно указать, что предлагаемый вариант должен быть выбран, если нижний предел оказывается строго меньше подставляемого значения. Вместо знака # можно также указывать знак ≤ (`\u2264` в Юникоде). По желанию можно даже указать для первого значения нижний предел равным -∞ (`-\u221E` в Юникоде), как показано ниже.

```
-∞<no houses|0<one house|2≤{1} houses
```

Непосредственно в Юникоде это же выражение будет выглядеть следующим образом:

```
-\u221E<no houses|0<one house|2\u2264{1} houses
```

В завершение примера форматирования текстового сообщения о последствиях стихийного бедствия разместим строку с условиями выбора в исходной строке сообщения. В результате получится следующий шаблон форматирования на английском языке:

```
String pattern = "On {2,date,long}, {0} destroyed
          {1,choice,0#no houses|1#one house|2#{1} houses}"
       + "and caused {3,number,currency} of damage.";
```

Для форматирования на немецком языке этот шаблон будет выглядеть следующим образом:

```
String pattern = "{0} zerstörte am {2,date,long}
          {1,choice,0#kein Haus|1#ein Haus|2#{1} Häuser}"
       + "und richtete einen Schaden von
             {3,number,currency} an.";
```

Примечательно, что порядок слов в шаблонах форматирования на английском и немецком языках разный, но методу `format()` передается *тот же самый* массив объектов. Под требуемый порядок слов в форматирующей строке подстраивается только последовательность заполнителей.

7.6. Ввод-вывод текста

Как вам должно быть уже известно, кодирование символов в Java основывается на Юникоде. Но в операционных системах Windows и Mac OS X до сих пор применяются устаревшие кодировки символов, часто несовместимые с другими, например, Windows-1252 и Mac Roman в странах Западной Европы или BIG5 на Тайване. Поэтому организовать взаимодействие с пользователями через текст оказывается не так просто, как может показаться на первый взгляд. Трудности, возникающие на этом пути, рассматриваются в последующих разделах.

7.6.1. Текстовые файлы

В настоящее время для сохранения или загрузки текстовых файлов лучше всего пользоваться кодировкой UTF-8, хотя может возникнуть потребность в обработке текстовых файлов с устаревшей кодировкой. Если кодировка символов известна заранее, ее можно указать при записи или чтении текстовых файлов, как показано ниже.

```
var out = new PrintWriter(filename, "Windows-1252");
```

Чтобы выяснить наиболее подходящую кодировку, можно получить "платформенную" кодировку, сделав следующий вызов:

```
Charset platformEncoding = Charset.defaultCharset();
```

7.6.2. Окончания строк

Правильная интерпретация окончаний строк — дело не региональных настроек, а платформ. Так, в Windows предполагается обнаружить последовательность символов `\r\n` в конце каждой строки текстового файла, тогда как в UNIX-подобных системах в конце строки достаточно указать последовательность символов `\n`. Впрочем, большинство современных прикладных программ для Windows

способно правильно интерпретировать и последовательность символов \n. Примечательным исключением из этого правила служит текстовый редактор Notepad, и если текстовых файл, выбираемый двойным щелчком мышью на его имени, требуется автоматически загрузить в текстовый редактор Notepad, следует обеспечить правильное окончание строк в нем.

Любая строка, выводимая с помощью метода println(), получает надлежащее окончание. Единственное затруднение возникает при выводе символьных строк, оканчивающихся последовательностью символов \n, поскольку они не преобразуются автоматически в окончания строк, принятые на конкретной платформе.

Вместо употребления последовательности символов \n для окончаний строк можно вызвать метод printf(), указав спецификатор формата %n для получения платформенно-ориентированных окончаний строк. Например, в результате следующего вызова:

```
out.printf("Hello%nWorld%n");
```

в Windows получается такая строка:

```
Hello\r\nWorld\r\n
```

а в других операционных системах — приведенная ниже строка.

```
Hello\nWorld\n
```

7.6.3. Консольный ввод-вывод

При написании прикладных программ, взаимодействующих с пользователями через стандартный ввод-вывод (объекты System.in/System.out) или консоль (метод System.console()), приходится принимать во внимание, что на консоли может использоваться иная, чем платформенная кодировка, о которой сообщает метод Charset.defaultCharset(). Эта особенность оказывается заметной при переходе к командной оболочке cmd в Windows. В версии этой оболочки для США применяется архаичная кодировка IBM437, внедренная на ПК IBM еще в 1982 году. Для обнаружения подобной информации отсутствует официально утвержденный прикладной программный интерфейс API. Например, знак денежной единицы евро (€) имеет свое представление в кодировке Windows-1252, тогда как в кодировке IBM437 такое представление отсутствует. Поэтому в результате вызова

```
System.out.println("100 €");
```

на консоль выводится такой результат:

```
100 ?
```

Пользователям прикладной программы можно порекомендовать сменить кодировку символов на консоли. В Windows для этого можно воспользоваться командой chcp. Например, по следующей команде:

```
chcp 1252
```

консоль перейдет к кодовой странице Windows-1252.

В идеальном случае пользователи должны перевести консоль на кодировку UTF-8. С этой целью в Windows можно выполнить следующую команду:

```
chcp 65001
```

Но, к сожалению, этого оказывается недостаточно, чтобы на консоли в Java применялась кодировка UTF-8. Ее необходимо также установить неофициально в системном свойстве `file.encoding` по следующей команде:

```
java -Dfile.encoding=UTF-8 МояПрограмма
```

7.6.4. Протокольные файлы

Когда протокольные сообщения направляются из библиотеки `java.util.logging` на консоль, они выводятся в кодировке, принятой на консоли. О том, как управлять этим процессом, упоминалось в предыдущем разделе. Но для вывода протокольных сообщений в файл служит класс `FileHandler`, где по умолчанию применяется платформенная кодировка.

Чтобы перейти к кодировке UTF-8, необходимо внести изменения в настройки диспетчера протоколирования. С этой целью в файле конфигурации протоколирования делается следующая установка:

```
java.util.logging.FileHandler.encoding=UTF-8
```

7.6.5. Отметка порядка следования байтов в кодировке UTF-8

Как упоминалось ранее, для обработки текстовых файлов рекомендуется применять кодировку UTF-8 при всякой возможности. Так, при чтении в прикладной программе текстовых файлов, созданных в других программах, может возникнуть еще одно затруднение в связи с тем, то в файл вполне допускается вводить символ, обозначающий отметку порядка следования байтов (`U+FEFF`). В кодировке UTF-16, где каждая кодовая единица представлена двумя байтами, такая отметка сообщает читающей файл программе, что в нем применяется порядок следования байтов от старшего к младшему или же от младшего к старшему. В кодировке UTF-8 каждая кодовая единица представлена одним байтом, поэтому указывать порядок следования байтов в этом случае не требуется. Но если файл начинается с байтов `0xEF 0xBB 0xBF`, что соответствует коду символа `U+FEFF` в кодировке UTF-8, то это явно свидетельствует о применении кодировки UTF-8. Именно такая практика и рекомендуется в стандарте Unicode, когда любая читающая файл программа отбрасывает первоначальную отметку порядка следования байтов.

Такой практике присущ лишь один недостаток. Компания Oracle в своей реализации языка Java упорно отказывается следовать стандарту Unicode, ссылаясь на потенциальную несовместимость. А это означает, что программирующие на Java должны сделать то, что не сделает платформа, а именно: проигнорировать код символа `U+FEFF`, если он встретится вначале читаемого текстового файла.

ВНИМАНИЕ! К сожалению, разработчики комплекта JDK не следуют приведенной выше рекомендации. Если передать компилятору **`javac`** достоверный исходный файл в кодировке UTF-8, который начинается с отметки порядка следования байтов, его компиляция завершится неудачно выдачей сообщения об ошибке **`"illegal character: \65279"`** (недопустимый символ: код \65279).

7.6.6. Кодирование символов в исходных файлах

Не следует забывать, что при написании программы на Java неизбежно приходится иметь дело с компилятором, пользуясь инструментальными средствами в *локальной* операционной системе. Допустим, для создания исходных файлов программы на Java используется стандартный текстовый редактор Notepad в китайской версии Windows. Полученный в итоге исходный код *не* является переносимым (т.е. независимым от платформы) из-за того, что в нем используется локальная кодировка символов (GB или BIG5, в зависимости от региональных настроек операционной системы). Классы становятся переносимыми только после компиляции, и в этом случае идентификаторы и текстовые сообщения представляются с помощью модифицированной кодировки UTF-8. Это означает, что при компиляции и выполнении программы предполагается использование трех перечисленных ниже кодировок символов.

- Исходные файлы: локальная кодировка.
- Файлы классов: модифицированная кодировка UTF-8.
- Виртуальная машина: кодировка UTF-16.

(О модифицированной кодировке UTF-8 и кодировке UTF-16 см. в главе 1.)

СОВЕТ. Конкретную кодировку исходного файла можно указать в командной строке с помощью параметра `-encoding`:

```
javac -encoding Big5 Myfile.java
```

7.7. Комплекты ресурсов

При интернационализации приложений на другие языки приходится переводить огромное количество сообщений, надписей на кнопках и т.п. Для упрощения этой задачи рекомендуется собрать все переводимые символьные строки в отдельном месте, которое обычно называется *ресурсом*. В этом случае переводчику достаточно отредактировать файлы ресурсов, не затрагивая исходный код программы. В языке Java для определения строковых ресурсов используются файлы свойств, а для других разновидностей ресурсов создаются классы ресурсов.

НА ЗАМЕТКУ! Технология использования ресурсов в Java отличается от аналогичной технологии в операционных системах Windows и Mac OS. В программе, предназначенной для выполнения под Windows или Mac OS, такие ресурсы, как меню, диалоговые окна, пиктограммы и сообщения, хранятся отдельно от самой программы. Поэтому специальный редактор ресурсов позволяет просматривать и видоизменять эти ресурсы, не затрагивая программный код.

НА ЗАМЕТКУ! В главе 5 первого тома настоящего издания описывается принцип размещения ресурсов (файлов данных, звука и изображения) в архивном JAR-файле. Метод **`getResource()`** из класса **`Class`** находит нужный файл, открывает его и возвращает URL, указывающий на искомый ресурс. При размещении файлов в архивном JAR-файле поиск файлов возлагается на загрузчик классов. Однако этот механизм не поддерживает региональные настройки.

7.7.1. Обнаружение комплектов ресурсов

Для интернационализации приложений создаются так называемые *комплекты ресурсов*. Каждый комплект представляет собой файл свойств или класс, который описывает элементы, характерные для конкретных региональных настроек, например сообщения, надписи и т.д. Для каждого комплекта ресурсов должны быть предоставлены все региональные настройки, поддержка которых предусматривается в прикладной программе.

Комплекты ресурсов именуются по специальным условным обозначениям. Например, ресурсы, характерные для Германии, размещаются в файле *имяКомплекта*_de_DE, а ресурсы, общие для стран, в которых используется немецкий язык, — в классе *имяКомплекта*_de. В целом комплекты ресурсов для отдельных стран именуются следующим образом:

имяКомплекта_язык_страна

Комплекты ресурсов для всех одноязычных стран именуются таким образом:

имяКомплекта_язык

Наконец, в качестве резерва ресурсы, применяемые по умолчанию, размещаются в файле, имя которого не указывается без всяких суффиксов. Загружается комплект ресурсов следующим образом:

```
ResourceBundle currentResources =
        ResourceBundle.getBundle(имяКомплекта,
                                 текущиеРегиональныеНастройки);
```

Метод `getBundle()` пытается загрузить комплект ресурсов, совпадающий с текущими региональными настройками по языку и стране. Если попытка загрузки завершится неудачей, то поочередно опускается страна и язык. Затем аналогичный поиск ресурсов осуществляется в региональных настройках по умолчанию и происходит обращение к комплекту ресурсов, выбираемому по умолчанию. Если и эта попытка оказывается безуспешной, то генерируется исключение типа `MissingResourceException`.

Таким образом, метод `getBundle()` пытается загрузить комплекты ресурсов в приведенной ниже последовательности, где *ТРН* — текущие региональные настройки, а *РНУ* — региональные настройки по умолчанию.

имяКомплекта_языкТРН_страна ТРН
имяКомплекта_языкТРН
имяКомплекта_языкТРН_страна РНУ
имяКомплекта_языкРНУ
имяКомплекта

И даже после того, как метод `getBundle()` обнаружит комплект ресурсов, например *имяКомплекта*_de_DE, он продолжит поиск ресурсов в комплектах *имяКомплекта*_de и *имяКомплекта*. Если эти комплекты ресурсов существуют, они становятся *родительскими* по отношению к комплекту *имяКомплекта*_de_DE в *иерархии ресурсов*. Родительские комплекты требуются на тот случай, если нужный ресурс не удастся обнаружить в текущем комплекте. Иными словами, если нужный ресурс не найден в комплекте *имяКомплекта*_de_DE, его поиск продолжается в комплектах *имяКомплекта*_de и *имяКомплекта*.

Очевидно, что это очень полезный механизм, но для его реализации пришлось бы немало программировать вручную. Поэтому механизм поддержки комплектов ресурсов в Java автоматически находит ресурсы, в наибольшей степени соответствующие конкретным региональным настройкам. Чтобы ввести в существующую программу новые региональные настройки, достаточно дополнить ими соответствующие комплекты ресурсов.

НА ЗАМЕТКУ! Обнаружение комплектов ресурсов рассматривается здесь в упрощенном виде. Если региональные настройки содержат письмо или вариант языка, поиск ресурсов значительно усложняется. Подробнее об этом можно узнать из документации на метод **`ResourceBundle.Control.getCandidateLocales()`**.

СОВЕТ. При разработке прикладной программы совсем не обязательно размещать все ресурсы в одном комплекте. Вместо этого один комплект ресурсов можно создать для надписей на кнопках, другой — для сообщений об ошибках и т.д.

7.7.2. Файлы свойств

Интернационализация символьных строк осуществляется довольно просто. Для этого достаточно разместить все символьные строки в файле свойств, например `MyProgramStrings.properties`. Файл свойств представляет собой обычный текстовый файл, каждая строка которого содержит ключ и значение, как показано ниже.

```
computeButton=Rechnen
colorName=black
defaultPaperSize=210´297
```

Файлы свойств именуются по принципу, описанному в предыдущем разделе, например, следующим образом:

```
MyProgramStrings.properties
MyProgramStrings_en.properties
MyProgramStrings_de_DE.properties
```

Комплект ресурсов можно загрузить аналогично приведенному ниже.

```
ResourceBundle bundle = ResourceBundle.getBundle(
                        "MyProgramStrings", locale);
```

Для поиска конкретной символьной строки потребуется вызов, подобный следующему:

```
String computeButtonLabel =
       bundle.getString("computeButton");
```

ВНИМАНИЕ! До версии Java 9 файлы свойств должны были содержать символы только в коде ASCII. В прежних версиях Java для размещения в этих файлах символов в Юникоде следует использовать формат **`\uxxxx`** Например, строка **`colorName=Grün`** будет иметь следующий вид:

```
colorName=Gr\u00FCn
```

Для подобного преобразования символов в файлах свойств можно также воспользоваться упоминавшейся ранее утилитой **`native2ascii`**.

7.7.3. Классы комплектов ресурсов

Для поддержки ресурсов, не являющихся символьными строками, следует определить классы, производные от класса `ResourceBundle`. Выбор имен для таких классов осуществляется в соответствии со стандартными обозначениями, как показано в приведенном ниже примере.

```
MyProgramResources.java
MyProgramResources_en.java
MyProgramResources_de_DE.java
```

Для загрузки класса комплекта ресурсов применяется тот же метод `getBundle()`, что и для загрузки файла свойств:

```
ResourceBundle bundle = ResourceBundle.getBundle(
                        "MyProgramResources", locale);
```


ВНИМАНИЕ! Если два комплекта ресурсов, один из которых реализован в виде класса, а другой — в виде файла свойств, имеют одинаковые имена, то при загрузке предпочтение отдается классу.

В каждом классе комплекта ресурсов реализуется таблица поиска. Для получения каждого интернационализируемого значения следует предоставить символьную строку с соответствующим ключом, как показано ниже.

```
var backgroundColor = (Color) bundle.getObject(
                                "backgroundColor");
double[] paperSize = (double[]) bundle.getObject(
                                "defaultPaperSize");
```

Самый простой способ реализовать класс комплекта ресурсов — создать подкласс, производный от класса `ListResourceBundle`. Класс `ListResourceBundle` позволяет разместить сначала все ресурсы в массиве объектов, а затем выполнить их поиск. Общая форма реализации подкласса, производного от класса `ListResourceBundle`, выглядит следующим образом:

```
public class имяКомплекта_язык_страна
       extends ListResourceBundle
{
  private static final Object[][] contents =
  {
    { ключ1, значение1 },
    { ключ2, значение2 },
    . . .
  }
  public Object[][] getContents() { return contents; }
}
```

Ниже приведены некоторые примеры реализации подклассов, производных от класса `ListResourceBundle`.

```
public class ProgramResources_de extends ListResourceBundle
{
  private static final Object[][] contents =
  {
    { "backgroundColor", Color.black },
```

```
    { "defaultPaperSize", new double[] { 210, 297 } }
  }
  public Object[][] getContents() { return contents; }
}

public class ProgramResources_en_US
       extends ListResourceBundle
{
  private static final Object[][] contents =
  {
    { "backgroundColor", Color.blue },
    { "defaultPaperSize", new double[] { 216, 279 } }
  }
public Object[][] getContents() { return contents; }
}
```

НА ЗАМЕТКУ! Размеры стандартных форматов бумаги задаются в миллиметрах. Во всех странах мира, кроме США и Канады, используются форматы бумаги, размеры которых определяются по стандарту ISO 216 (дополнительные сведения по данному вопросу можно найти в документе, доступном по адресу **https://www.cl.cam.ac.uk/%7Emgk25/iso-paper.html**).

Класс комплекта ресурсов можно также создать как подкласс, производный от класса `ResourceBundle`. В этом случае придется реализовать два приведенных ниже метода для получения объекта типа `Enumeration`, содержащего ключи, а также для извлечения конкретного значения по заданному ключу. Метод `getObject()` из класса `ResourceBundle` вызывает определяемый пользователем метод `handleGetObject()`.

```
Enumeration<String> getKeys()
Object handleGetObject(String key)
```

java.util.ResourceBundle 1.1

- **static ResourceBundle getBundle(String *baseName*, Locale *loc*)**
- **static ResourceBundle getBundle(String *baseName*)**

  Загружают класс комплекта ресурсов по указанному имени, а также его родительские классы в соответствии с заданными или устанавливаемыми по умолчанию региональными настройками. Если классы комплектов ресурсов находятся в пакете, то должно быть указано полное имя такого класса, например **intl.Programresources**. Классы комплектов ресурсов должны быть объявлены открытыми (**public**), чтобы сделать их доступными для метода **getBundle()**.

- **Object getObject(String *name*)**

  Извлекает объект из указанного комплекта ресурсов или его родительских комплектов.

- **String getString(String *name*)**

  Извлекает объект из указанного комплекта ресурсов или его родительских комплектов и приводит его к типу **String**.

- **String[] getStringArray(String *name*)**

  Извлекает объект из комплекта ресурсов или его родительских комплектов и представляет его в виде массива символьных строк.

java.util.ResourceBundle 1.1 *(окончание)*

- **Enumeration<String> getKeys()**

  Возвращает объект перечисления типа **Enumeration**, содержащий ключи из текущего комплекта ресурсов. В этом объекте перечисляются также ключи из родительских комплектов ресурсов.

- **Object handleGetObject(String *key*)**

  Если реализуется собственный механизм поиска ресурсов, этот метод следует переопределить таким образом, чтобы он возвращал значение по заданному ключу.

7.8. Пример интернационализации прикладной программы

В этом разделе изложенный выше материал применяется на практике для интернационализации прикладной программы для расчета пенсионных сбережений. В этой программе определяется, достаточно ли отчисляется денежных средств для выхода на пенсию. Для этого следует ввести свой возраст, сумму, которая отчисляется каждый месяц, а также указать другие данные (рис. 7.4)

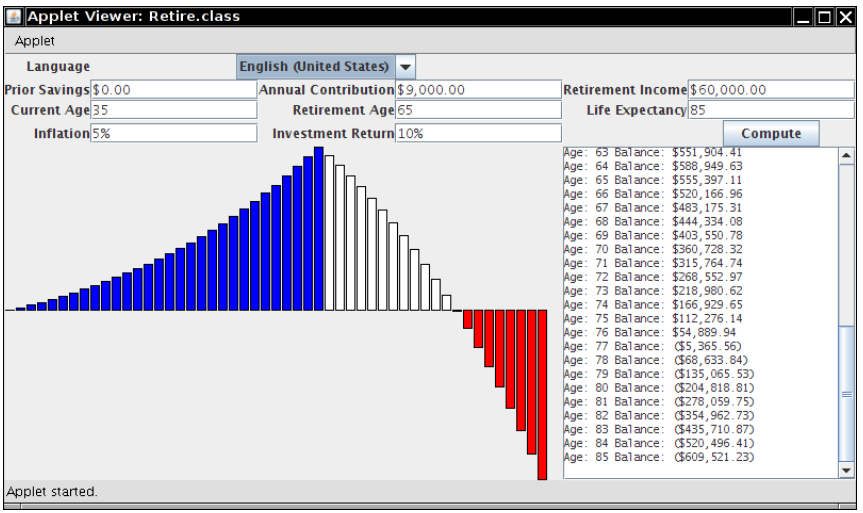


Рис. 7.4. Рабочее окно программы для расчета пенсионных сбережений с пользовательским интерфейсом на английском языке

В текстовой области и на графике представлены данные о ежегодном состоянии пенсионного счета. Если с возрастом остатки на пенсионном счете становятся отрицательными, что наглядно показано на части графика, построенной ниже оси абсцисс, в таком случае следует принять какие-то меры, например, строже экономить деньги, отложить дату выхода на пенсию и т.п.

Программа для расчета пенсионных сбережений имеет три варианта пользовательского интерфейса на английском, немецком и китайском языках. Ниже перечислены основные особенности интернационализации данной прикладной программы.

- Надписи и сообщения переводятся с английского на немецкий и китайский языки. Соответствующие ресурсы находятся в классах `RetireResources_de` и `RetireResources_zh`, а ресурсы для пользовательского интерфейса на английском языке используются в качестве резервных и находятся в классе `RetireResources`.
- При изменении региональных настроек заново форматируются надписи и содержимое текстовых полей.
- Содержимое текстовых полей для ввода чисел, денежных сумм и процентов представляется в формате, соответствующем выбранным региональным настройкам.
- Для форматирования расчетного поля используется класс `MessageFomat`. Форматирующая строка хранится в комплекте ресурсов для каждого языка.
- Цвет графика меняется в зависимости от выбранного языка. Это никак не влияет на выполнение программой ее функций, а лишь демонстрирует подобную возможность.

В листингах 7.5–7.8 приведен исходный код рассматриваемой здесь прикладной программы, а в листингах 7.9–7.11 — содержимое файлов свойств для интернационализированных строк. На рис. 7.5 и 7.6 показаны немецкий и китайский варианты пользовательского интерфейса данной программы. Для работы с данной программой на китайском языке необходимо запустить ее в китайской версии Windows или установить самостоятельно китайские шрифты в операционной системе.

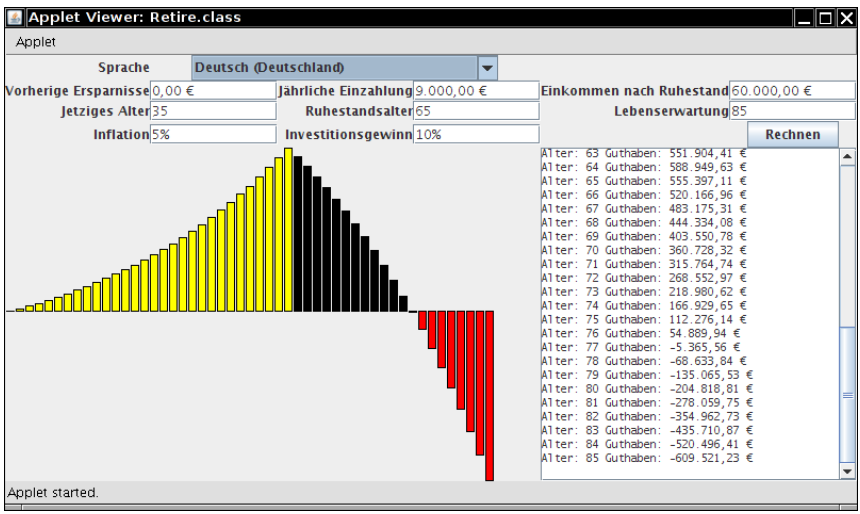


Рис. 7.5. Рабочее окно программы для расчета пенсионных сбережений с пользовательским интерфейсом на немецком языке

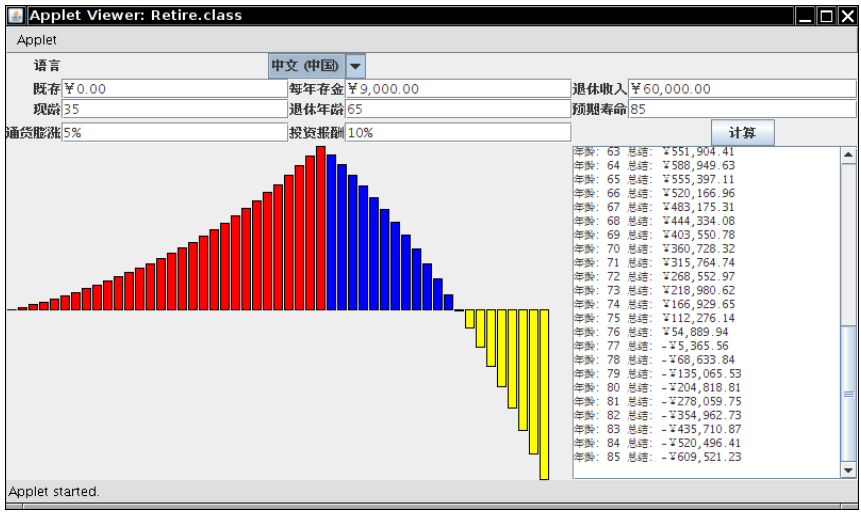


Рис. 7.6. Рабочее окно программы для расчета пенсионных сбережений с пользовательским интерфейсом на китайском языке

Listing 7.5 retire/Retire.java

```
1  package retire;
2
3  import java.awt.*;
4  import java.awt.geom.*;
5  import java.text.*;
6  import java.util.*;
7
8  import javax.swing.*;
9
10 /**
11  * This program shows a retirement calculator. The UI is displayed in English, German, and
12  * Chinese.
13  * @version 1.25 2018-05-01
14  * @author Cay Horstmann
15  */
16 public class Retire
17 {
18    public static void main(String[] args)
19    {
20       EventQueue.invokeLater(() ->
21          {
22             var frame = new RetireFrame();
23             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24             frame.setVisible(true);
25          });
26    }
27 }
28
```

```
32  class RetireFrame extends JFrame
33  {
34    private JTextField savingsField = new JTextField(10);
35    private JTextField contribField = new JTextField(10);
36    private JTextField incomeField = new JTextField(10);
37    private JTextField currentAgeField =
38                       new JTextField(4);
39    private JTextField retireAgeField =
40                       new JTextField(4);
41    private JTextField deathAgeField = new JTextField(4);
42    private JTextField inflationPercentField =
43                       new JTextField(6);
44    private JTextField investPercentField =
45                       new JTextField(6);
46    private JTextArea retireText = new JTextArea(10, 25);
47    private RetireComponent retireCanvas =
48                       new RetireComponent();
49    private JButton computeButton = new JButton();
50    private JLabel languageLabel = new JLabel();
51    private JLabel savingsLabel = new JLabel();
52    private JLabel contribLabel = new JLabel();
53    private JLabel incomeLabel = new JLabel();
54    private JLabel currentAgeLabel = new JLabel();
55    private JLabel retireAgeLabel = new JLabel();
56    private JLabel deathAgeLabel = new JLabel();
57    private JLabel inflationPercentLabel = new JLabel();
58    private JLabel investPercentLabel = new JLabel();
59    private RetireInfo info = new RetireInfo();
60    private Locale[] locales =
61            { Locale.US, Locale.CHINA, Locale.GERMANY };
62    private Locale currentLocale;
63    private JComboBox<Locale> localeCombo =
64           new LocaleCombo(locales);
65    private ResourceBundle res;
66    private ResourceBundle resStrings;
67    private NumberFormat currencyFmt;
68    private NumberFormat numberFmt;
69    private NumberFormat percentFmt;
70
71    public RetireFrame()
72    {
73      setLayout(new GridBagLayout());
74      add(languageLabel, new GBC(0, 0)
75         .setAnchor(GBC.EAST));
76      add(savingsLabel, new GBC(0, 1)
77         .setAnchor(GBC.EAST));
78      add(contribLabel, new GBC(2, 1)
79         .setAnchor(GBC.EAST));
80      add(incomeLabel, new GBC(4, 1)
81         .setAnchor(GBC.EAST));
82      add(currentAgeLabel, new GBC(0, 2)
83         .setAnchor(GBC.EAST));
84      add(retireAgeLabel, new GBC(2, 2)
85         .setAnchor(GBC.EAST));
86      add(deathAgeLabel, new GBC(4, 2)
87         .setAnchor(GBC.EAST));
88      add(inflationPercentLabel, new GBC(0, 3)
```

```
89          .setAnchor(GBC.EAST));
90       add(investPercentLabel, new GBC(2, 3)
91          .setAnchor(GBC.EAST));
92       add(localeCombo, new GBC(1, 0, 3, 1));
93       add(savingsField, new GBC(1, 1).setWeight(100, 0)
94          .setFill(GBC.HORIZONTAL));
95       add(contribField, new GBC(3, 1).setWeight(100, 0)
96          .setFill(GBC.HORIZONTAL));
97       add(incomeField, new GBC(5, 1).setWeight(100, 0)
98          .setFill(GBC.HORIZONTAL));
99       add(currentAgeField, new GBC(1, 2).setWeight(100, 0)
100         .setFill(GBC.HORIZONTAL));
101      add(retireAgeField, new GBC(3, 2).setWeight(100, 0)
102         .setFill(GBC.HORIZONTAL));
103      add(deathAgeField, new GBC(5, 2).setWeight(100, 0)
104         .setFill(GBC.HORIZONTAL));
105      add(inflationPercentField, new GBC(1, 3)
106         .setWeight(100, 0).setFill(GBC.HORIZONTAL));
107      add(investPercentField, new GBC(3, 3)
108         .setWeight(100, 0).setFill(GBC.HORIZONTAL));
109      add(retireCanvas, new GBC(0, 4, 4, 1)
110         .setWeight(100, 100).setFill(GBC.BOTH));
111      add(new JScrollPane(retireText), new GBC(4, 4, 2, 1)
112         .setWeight(0, 100).setFill(GBC.BOTH));
113
114      computeButton.setName("computeButton");
115      computeButton.addActionListener(event ->
116        {
117          getInfo();
118          updateData();
119          updateGraph();
120        });
121      add(computeButton, new GBC(5, 3));
122
123      retireText.setEditable(false);
124      retireText.setFont(new Font(
125                "Monospaced", Font.PLAIN, 10));
126
127      info.setSavings(0);
128      info.setContrib(9000);
129      info.setIncome(60000);
130      info.setCurrentAge(35);
131      info.setRetireAge(65);
132      info.setDeathAge(85);
133      info.setInvestPercent(0.1);
134      info.setInflationPercent(0.05);
135
136      // региональные настройки США
137      // выбираются по умолчанию:
138      int localeIndex = 0;
139      for (int i = 0; i < locales.length; i++)
140        // если текущие региональные настройки относятся
141        // к числу выбираемых, то выбрать их:
142        if (getLocale().equals(locales[i]))
143          localeIndex = i;
144      setCurrentLocale(locales[localeIndex]);
145
```

```
146     localeCombo.addActionListener(event ->
147       {
148         setCurrentLocale((Locale)
149                   localeCombo.getSelectedItem());
150         validate();
151       });
152     pack();
153   }
154
155   /**
156    * Устанавливает текущие региональные настройки
157    * @param locale Требующиеся региональные настройки
158    */
159   public void setCurrentLocale(Locale locale)
160   {
161     currentLocale = locale;
162     localeCombo.setLocale(currentLocale);
163     localeCombo.setSelectedItem(currentLocale);
164
165     res = ResourceBundle.getBundle(
166           "retire.RetireResources", currentLocale);
167     resStrings = ResourceBundle.getBundle(
168                "retire.RetireStrings", currentLocale);
169     currencyFmt =
170        NumberFormat.getCurrencyInstance(currentLocale);
171     numberFmt =
172        NumberFormat.getNumberInstance(currentLocale);
173     percentFmt =
174        NumberFormat.getPercentInstance(currentLocale);
175
176     updateDisplay();
177     updateInfo();
178     updateData();
179     updateGraph();
180   }
181
182   /**
183    * Обновляет все метки при отображении
184    */
185   public void updateDisplay()
186   {
187     languageLabel.setText(
188                  resStrings.getString("language"));
189     savingsLabel.setText(
190                 resStrings.getString("savings"));
191     contribLabel.setText(
192                 resStrings.getString("contrib"));
193     incomeLabel.setText(
194                resStrings.getString("income"));
195     currentAgeLabel.setText(
196                resStrings.getString("currentAge"));
197     retireAgeLabel.setText(
198                   resStrings.getString("retireAge"));
200     deathAgeLabel.setText(
201                  resStrings.getString("deathAge"));
202     inflationPercentLabel.setText(
203              resStrings.getString("inflationPercent"));
```

```
204     investPercentLabel.setText(
205              resStrings.getString("investPercent"));
206     computeButton.setText(
207              resStrings.getString("computeButton"));
208   }
209
210   /**
211    * Обновляет данные в текстовых полях
212    */
213   public void updateInfo()
214   {
215     savingsField.setText(
216            currencyFmt.format(info.getSavings()));
217     contribField.setText(
218            currencyFmt.format(info.getContrib()));
219     incomeField.setText(
220           currencyFmt.format(info.getIncome()));
221     currentAgeField.setText(
222            numberFmt.format(info.getCurrentAge()));
223     retireAgeField.setText(
224           numberFmt.format(info.getRetireAge()));
225     deathAgeField.setText(
226           numberFmt.format(info.getDeathAge()));
227     investPercentField.setText(
228           percentFmt.format(info.getInvestPercent()));
229     inflationPercentField.setText(
230         percentFmt.format(info.getInflationPercent()));
231   }
232
233   /**
234    * Обновляет данные, отображаемые в текстовой области
235    */
236   public void updateData()
237   {
238     retireText.setText("");
239     var retireMsg = new MessageFormat("");
240     retireMsg.setLocale(currentLocale);
241     retireMsg.applyPattern(
242               resStrings.getString("retire"));
243
244     for (int i = info.getCurrentAge();
245          i <= info.getDeathAge(); i++)
246     {
247       Object[] args = { i, info.getBalance(i) };
248       retireText.append(retireMsg.format(args) + "\n");
249     }
250   }
251
252   /**
253    * Обновляет график
254    */
255   public void updateGraph()
256   {
257     retireCanvas.setColorPre((Color)
258                  res.getObject("colorPre"));
259     retireCanvas.setColorGain((Color)
260                  res.getObject("colorGain"));
```

```
261     retireCanvas.setColorLoss((Color)
262                 res.getObject("colorLoss"));
263     retireCanvas.setInfo(info);
264     repaint();
265   }
266
267   /**
268    * Считывает данные, вводимые пользователем
269    * в текстовых полях
270    */
271   public void getInfo()
272   {
273     try
274     {
275       info.setSavings(currencyFmt.parse(
276            savingsField.getText()).doubleValue());
277       info.setContrib(currencyFmt.parse(
278            contribField.getText()).doubleValue());
279       info.setIncome(currencyFmt.parse(
280            incomeField.getText()).doubleValue());
281       info.setCurrentAge(numberFmt.parse(
282            currentAgeField.getText()).intValue());
283       info.setRetireAge(numberFmt.parse(
284            retireAgeField.getText()).intValue());
285       info.setDeathAge(numberFmt.parse(
286            deathAgeField.getText()).intValue());
287
288       info.setInvestPercent(percentFmt.parse(
289           investPercentField.getText()).doubleValue());
290       info.setInflationPercent(percentFmt.parse(
291                       inflationPercentField
292                      .getText()).doubleValue());
293     }
294     catch (ParseException ex)
295     {
296       ex.printStackTrace();
297     }
298   }
299 }
300
301 /**
302  * Данные, требуемые для расчета пенсионных отчислений
303  */
304 class RetireInfo
305 {
306   private double savings;
307   private double contrib;
308   private double income;
309   private int currentAge;
310   private int retireAge;
311   private int deathAge;
312   private double inflationPercent;
313   private double investPercent;
314   private int age;
315   private double balance;
316
317   /**
```

```
318    * Получает остаток на счете, имеющийся
319    * на указанный год
320    * @param year Год для расчета остатка на счете
321    * @return Сумма, имеющаяся (или требующаяся)
322    *         на указанный год
323    */
324   public double getBalance(int year)
325   {
326     if (year < currentAge) return 0;
327     else if (year == currentAge)
328     {
329       age = year;
330       balance = savings;
331       return balance;
332     }
333     else if (year == age) return balance;
334     if (year != age + 1) getBalance(year - 1);
335     age = year;
336     if (age < retireAge) balance += contrib;
337     else balance -= income;
338     balance = balance * (1 + (investPercent
339                               - inflationPercent));
340     return balance;
341   }
342
343   /**
344    * Получает сумму предыдущих сбережений
345    * @return Сумма сбережений
346    */
347   public double getSavings()
348   {
349     return savings;
350   }
351
352   /**
353    * Устанавливает сумму предыдущих сбережений
354    * @param newValue Сумма сбережений
355    */
356   public void setSavings(double newValue)
357   {
358     savings = newValue;
359   }
360
361   /**
362    * Получает сумму ежегодных отчислений
363    * на пенсионный счет
364    * @return Сумма ежегодных отчислений
365    */
366   public double getContrib()
367   {
368     return contrib;
369   }
370
371   /**
372    * Устанавливает сумму ежегодных отчислений
373    * на пенсионный счет
374    * @param newValue Сумма ежегодных отчислений
```

```
375    */
376   public void setContrib(double newValue)
377   {
378     contrib = newValue;
379   }
380
381   /**
382    * Получает сумму ежегодного дохода
383    * @return Сумма ежегодного дохода
384    */
385   public double getIncome()
386   {
387     return income;
388   }
389
390   /**
391    * Устанавливает сумму ежегодного дохода
392    * @param newValue Сумма ежегодного дохода
393    */
394   public void setIncome(double newValue)
395   {
396     income = newValue;
397   }
398
399   /**
400    * Получает текущий возраст
401    * @return Текущий возраст
402    */
403   public int getCurrentAge()
404   {
405     return currentAge;
406   }
407
408   /**
409    * Устанавливает текущий возраст
410    * @param newValue Текущий возраст
411    */
412   public void setCurrentAge(int newValue)
413   {
414     currentAge = newValue;
415   }
416
417   /**
418    * Получает возраст для выхода на пенсию
419    * @return Пенсионный возраст
420    */
421   public int getRetireAge()
422   {
423     return retireAge;
424   }
425
426   /**
427    * Устанавливает возраст для выхода на пенсию
428    * @param newValue Пенсионный возраст
429    */
430   public void setRetireAge(int newValue)
431   {
```

```
432     retireAge = newValue;
433   }
434
435   /**
436    * Получает предполагаемую продолжительность жизни
437    * @return Предполагаемая продолжительность жизни
438    */
439   public int getDeathAge()
440   {
441     return deathAge;
442   }
443
444   /**
445    * Устанавливает предполагаемую
446    * продолжительность жизни
447    * @param newValue Предполагаемая продолжительность
448    *                 жизни
449    */
450   public void setDeathAge(int newValue)
451   {
452     deathAge = newValue;
453   }
454
455   /**
456    * Получает предполагаемый уровень
457    * инфляции в процентах
458    * @return Уровень инфляции в процентах
459    */
460   public double getInflationPercent()
461   {
462     return inflationPercent;
463   }
464
465   /**
466    * Устанавливает предполагаемый уровень
467    * инфляции в процентах
468    * @param newValue Уровень инфляции в процентах
469    */
470   public void setInflationPercent(double newValue)
471   {
472     inflationPercent = newValue;
473   }
474
475   /**
476    * Получает предполагаемый доход от капиталовложений
477    * @return Доход от капиталовложений в процентах
478    */
479   public double getInvestPercent()
480   {
481     return investPercent;
482   }
483
484   /**
485    * Устанавливает предполагаемый доход
486    * от капиталовложений
487    * @param newValue Доход от капиталовложений
488    *                 в процентах
```

```
489    */
490   public void setInvestPercent(double newValue)
491   {
492     investPercent = newValue;
493   }
494 }
495
496 /**
497  * Этот компонент рисует график результатов
498  * пенсионных вложений
499  */
500 class RetireComponent extends JComponent
501 {
502   private static final int PANEL_WIDTH = 400;
503   private static final int PANEL_HEIGHT = 200;
504   private static final Dimension PREFERRED_SIZE =
505           new Dimension(800, 600);
506   private RetireInfo info = null;
507   private Color colorPre;
508   private Color colorGain;
509   private Color colorLoss;
510
511   public RetireComponent()
512   {
513     setSize(PANEL_WIDTH, PANEL_HEIGHT);
514   }
515
516   /**
517    * Устанавливает данные для построения графика
518    * пенсионных вложений
519    * @param newInfo Новые данные о пенсионных вложениях
520    */
521   public void setInfo(RetireInfo newInfo)
522   {
523     info = newInfo;
524     repaint();
525   }
526
527   public void paintComponent(Graphics g)
528   {
529     var g2 = (Graphics2D) g;
530     if (info == null) return;
531
532     double minValue = 0;
533     double maxValue = 0;
534     int i;
535     for (i = info.getCurrentAge();
536          i <= info.getDeathAge(); i++)
537     {
538       double v = info.getBalance(i);
539       if (minValue > v) minValue = v;
540       if (maxValue < v) maxValue = v;
541     }
542     if (maxValue == minValue) return;
543
544     int barWidth = getWidth() / (info.getDeathAge()
545                    - info.getCurrentAge() + 1);
```

```
546     double scale = getHeight() / (maxValue - minValue);
547
548     for (i = info.getCurrentAge();
549          i <= info.getDeathAge(); i++)
550     {
551       int x1 = (i - info.getCurrentAge())
552                 * barWidth + 1;
553       int y1;
554       double v = info.getBalance(i);
555       int height;
556       int yOrigin = (int) (maxValue * scale);
557
558       if (v >= 0)
559       {
560         y1 = (int) ((maxValue - v) * scale);
561         height = yOrigin - y1;
562       }
563       else
564       {
565         y1 = yOrigin;
566         height = (int) (-v * scale);
567       }
568
569       if (i < info.getRetireAge())
570         g2.setPaint(colorPre);
571       else if (v >= 0) g2.setPaint(colorGain);
572       else g2.setPaint(colorLoss);
573       var bar = new Rectangle2D.Double(x1, y1,
574                     barWidth - 2, height);
575       g2.fill(bar);
576       g2.setPaint(Color.black);
577       g2.draw(bar);
578     }
579   }
580
581   /**
582    * Устанавливает цвет графика для периода
583    * до выхода на пенсию
584    * @param color the desired color
585    */
586   public void setColorPre(Color color)
587   {
588     colorPre = color;
589      repaint();
590   }
591
592   /**
593    * Устанавливает цвет графика для периода после
594    * выхода на пенсию, когда остаток на пенсионном
595    * счете еще положительный
596    * @param color Требующийся цвет
597   */
598   public void setColorGain(Color color)
599   {
600     colorGain = color;
601     repaint();
602   }
```

```
603
604    /**
605     * Устанавливает цвет графика для периода после
606     * выхода на пенсию, когда остаток на пенсионном
607     * счете уже отрицательный
608     * @param color Требующийся цвет
609     */
610    public void setColorLoss(Color color)
611    {
612      colorLoss = color;
613      repaint();
614    }
615
616    public Dimension getPreferredSize()
617    { return PREFERRED_SIZE; }
618 }
```

Листинг 7.6. Исходный код из файла **retire/RetireResources.java**

```
1  package retire;
2
3  import java.awt.*;
4
5  /**
6   * Нестроковые ресурсы для пользовательского
7   * интерфейса на английском языке программы для
8   * расчета пенсионных сбережений
9   * @version 1.21 2001-08-27
10  * @author Cay Horstmann
11 */
12 public class RetireResources
13       extends java.util.ListResourceBundle
14 {
15   private static final Object[][] contents = {
16   // НАЧАЛО ИНТЕРНАЦИОНАЛИЗАЦИИ
17      { "colorPre", Color.blue },
18      { "colorGain", Color.white },
19      { "colorLoss", Color.red }
20   // КОНЕЦ ИНТЕРНАЦИОНАЛИЗАЦИИ
21   };
22
23   public Object[][] getContents()
24   {
25     return contents;
26   }
27 }
```

Листинг 7.7. Исходный код из файла **retire/RetireResources_de.java**

```
1  package retire;
2
3  import java.awt.*;
4
5  /**
```

```
6   * Нестроковые ресурсы для пользовательского
7   * интерфейса на немецком языке программы для
8   * расчета пенсионных сбережений
9   * @version 1.21 2001-08-27
10  * @author Cay Horstmann
11  */
12 public class RetireResources_de
13        extends java.util.ListResourceBundle
14 {
15   private static final Object[][] contents = {
16   // НАЧАЛО ИНТЕРНАЦИОНАЛИЗАЦИИ
17      { "colorPre", Color.yellow },
18      { "colorGain", Color.black },
19      { "colorLoss", Color.red }
20   // КОНЕЦ ИНТЕРНАЦИОНАЛИЗАЦИИ
21   };
22
23   public Object[][] getContents()
24   {
25     return contents;
26   }
27 }
```

Листинг 7.8. Исходный код из файла **retire/RetireResources_zh.java**

```
1  package retire;
2
3  import java.awt.*;
4
5  /**
6   * Нестроковые ресурсы для пользовательского
7   * интерфейса на китайском языке программы для
8   * расчета пенсионных сбережений
9   * @version 1.21 2001-08-27
10  * @author Cay Horstmann
11 */
12 public class RetireResources_zh
13        extends java.util.ListResourceBundle
14 {
15   private static final Object[][] contents = {
16   // НАЧАЛО ИНТЕРНАЦИОНАЛИЗАЦИИ
17      { "colorPre", Color.red },
18      { "colorGain", Color.blue },
19      { "colorLoss", Color.yellow }
20   // КОНЕЦ ИНТЕРНАЦИОНАЛИЗАЦИИ
21   };
22
23   public Object[][] getContents()
24   {
25     return contents;
26   }
27 }
```

Листинг 7.9. Содержимое файла свойств **retire/RetireStrings.properties**

```
1 language=Language
2 computeButton=Compute
3 savings=Prior Savings
4 contrib=Annual Contribution
5 income=Retirement Income
6 currentAge=Current Age
7 retireAge=Retirement Age
8 deathAge=Life Expectancy
9 inflationPercent=Inflation
10 investPercent=Investment Return
11 retire=Age: {0,number} Balance: {1,number,currency}
```

Листинг 7.10. Содержимое файла свойств **retire/RetireStrings_de.properties**

```
1 language=Sprache
2 computeButton=Rechnen
3 savings=Vorherige Ersparnisse
4 contrib=J\u00e4hrliche Einzahlung
5 income=Einkommen nach Ruhestand
6 currentAge=Jetziges Alter
7 retireAge=Ruhestandsalter
8 deathAge=Lebenserwartung
9 inflationPercent=Inflation
10 investPercent=Investitionsgewinn
11 retire=Alter: {0,number} Guthaben: {1,number,currency}
```

Листинг 7.11. Содержимое файла свойств **retire/RetireStrings_zh.properties**

```
1  language=语言
2  computeButton=计算
3  savings=既存
4  contrib=每年存金
5  income=退休收入
6  currentAge=现龄
7  retireAge=退休年龄
8  deathAge=预期寿命
9  inflationPercent=通货膨涨
10 investPercent=投资报酬
11 retire=年龄: {0,number} 总结: {1,number,currency}
```

Теперь вам должно быть понятно, как пользоваться средствами интернационализации в Java. В частности, для перевода текстовой информации на многие языки служат комплекты ресурсов, а средства форматирования и сортировки применяются для обработки текста с учетом региональных настроек. В следующей главе будут рассмотрены вопросы написания сценариев, компиляции и обработки аннотаций.

ГЛАВА 8

Написание сценариев, компиляция и обработка аннотаций

В этой главе...

- Написание сценариев для платформы Java
- Прикладной интерфейс API для компилятора
- Применение аннотаций
- Синтаксис аннотаций
- Стандартные аннотации
- Обработка аннотаций на уровне исходного кода
- Конструирование байт-кодов

В этой главе рассматриваются три методики обработки кода. Прикладной интерфейс API для сценариев позволяет вызывать код на языке сценариев таким же образом, как и в языке JavaScript или Groovy. Прикладной интерфейс API для компилятора дает возможность компилировать исходный код Java в самой прикладной программе, а обработчики аннотаций — обрабатывать файлы классов или исходного кода Java, содержащие аннотации. Из этой главы вы узнаете, что существует немало приложений для обработки аннотаций, начиная с простой диагностики и заканчивая так называемым "конструированием байт-кодов" — вставкой байт-кодов в файлы классов и даже выполнением программ.

8.1. Написание сценариев для платформы Java

Язык сценариев — это такой язык программирования, который позволяет избегать обычного цикла операций редактирования, компиляции, компоновки и выполнения благодаря интерпретации исходного текста программы во время выполнения. Языки сценариев обладают рядом следующих преимуществ.

- Быстрый цикл обработки, стимулирующий стремление к экспериментированию.
- Возможность изменять поведение выполняющейся программы.
- Возможность для пользователей специально настраивать программы.

С другой стороны, у большинства языков сценариев отсутствуют средства, необходимые для разработки сложных прикладных программ, включая строгий контроль типов, инкапсуляцию и модульность.

В связи с этим возникает соблазн объединить преимущества языков сценариев с преимуществами традиционных языков программирования. Именно это и позволяет сделать прикладной интерфейс API для сценариев на платформе Java. В частности, он предоставляет возможность вызывать из программы на Java сценарии, написанные на JavaScript, Groovy, Ruby и даже таких экзотических языках, как Scheme и Haskell. Например, в проекте Renjin (`www.renjin.org`) предоставляется реализованная в Java возможность программировать на языке R, который зачастую применяется в области статистического программирования. С этой целью используется интерпретатор из прикладного интерфейса API для выполнения сценариев.

В последующих разделах будет показано, как выбирается интерпретатор сценариев для конкретного языка, как выполняются сценарии и как пользоваться дополнительными преимуществами, которые дают некоторые интерпретаторы сценариев.

8.1.1. Получение интерпретатора сценариев

Интерпретатор сценариев — это, по существу, библиотека, позволяющая выполнять сценарии на конкретном языке. При запуске виртуальная машина обнаруживает все доступные интерпретаторы сценариев. Получить их перечень можно, создав объект типа `ScriptEngineManager` и вызвав метод `getEngineFactories()`. Далее у каждой фабрики интерпретаторов сценариев можно запросить сведения об именах поддерживаемых интерпретаторов, типах MIME и расширениях файлов. В табл. 8.1 приведены наиболее употребительные интерпретаторы сценариев и соответствующие им типы и расширения.

Таблица 8.1. Свойства фабрик интерпретаторов сценариев

Интерпретатор	Имена	Типы MIME	Расширения
Nashorn (входит в состав JDK)	nashorn, Nashorn, js, JS, JavaScript, javascript, ECMAScript, ecmascript	**application/javascript**, **application/ecmascript**, **text/javascript**, **text/ecmascript**	**js**
Groovy	groovy	Отсутствуют	**groovy**
Renjin	Renjin	**text/x-R**	**R**, **r**, **S**, **s**

Обычно требующийся интерпретатор сценариев известен и может запрашиваться по имени, типу MIME или расширению, как показано в приведенном ниже примере.

```
ScriptEngine engine = manager.getEngineByName("nashorn");
```

В версии Java 8 внедрен интерпретатор Nashorn сценариев на языке JavaScript, разработанный компанией Oracle. Предоставив необходимые архивные JAR-файлы по пути к соответствующим классам, можно дополнить перечень языков написания сценариев.

`javax.script.ScriptEngineManager` 6

- **`List<ScriptEngineFactory> getEngineFactories()`**

  Получает список всех обнаруженных фабрик интерпретаторов сценариев.
- **`ScriptEngine getEngineByName(String name)`**
- **`ScriptEngine getEngineByExtension(String extension)`**
- **`ScriptEngine getEngineByMimeType(String mimeType)`**

  Получают интерпретатор сценариев с заданным именем, расширением файла сценария или типом MIME.

`javax.script.ScriptEngineFactory` 6

- **`List<String> getNames()`**
- **`List<String> getExtensions()`**
- **`List<String> getMimeTypes()`**

  Получают имена, расширения файлов сценариев и типы MIME, по которым известна данная фабрика.

8.1.2. Выполнение сценариев и привязки

Получив интерпретатор, можно приступить к вызову сценария:

```
Object result = engine.eval(scriptString);
```

Если сценарий хранится в файле, необходимо открыть поток чтения типа `Reader` и сделать следующий вызов:

```
Object result = engine.eval(reader);
```

С помощью одного и того же интерпретатора можно вызвать целый ряд сценариев. Если какой-нибудь из сценариев содержит определения переменных, функций или классов, большинство интерпретаторов сценариев будет сохранять их для последующего использования. Так, в приведенном ниже примере кода возвращается значение `1729`.

```
engine.eval("n = 1728");
Object result = engine.eval("n + 1");
```

НА ЗАМЕТКУ! Чтобы выяснить, безопасно ли параллельное выполнение сценариев во многих потоках, достаточно сделать следующий вызов:

```
Object param = factory.getParameter("THREADING");
```

В итоге возвращается одно из перечисленных ниже значений.

- **null**: параллельное выполнение небезопасно.
- **"MULTITHREADED"**: параллельное выполнение безопасно. Результаты исполнения одного потока могут быть доступны для другого потока.
- **"THREAD-ISOLATED"**: то же, что и значение **"MULTITHREADED"**, но только для каждого потока исполнения поддерживаются разные привязки переменных.
- **"STATELESS"**: то же, что и значение **"THREAD-ISOLATED"**, но только сценарии не могут изменять привязки переменных.

Интерпретатор сценариев нередко требуется дополнять привязками переменных. Каждая привязка состоит из имени и связываемого объекта Java. Рассмотрим в качестве примера следующие операторы:

```
engine.put(k, 1728);
Object result = engine.eval("k + 1");
```

Код сценария читает определение объекта `k` из привязок в области видимости интерпретатора. И это очень важно, так как почти все языки сценариев могут получать доступ к объектам Java и зачастую посредством более простого, чем у Java, синтаксиса. Например:

```
engine.put(b, new JButton());
engine.eval("b.text = 'Ok'");
```

С другой стороны, можно извлекать значения переменных, привязанных операторами сценария, как показано ниже.

```
engine.eval("n = 1728");
Object result = engine.get("n");
```

Кроме области видимости интерпретатора сценариев, существует и глобальная область видимости. Любые привязки, которые вводятся в объект типа `ScriptEngineManager`, становятся видимыми для всех интерпретаторов сценариев.

Вместо того чтобы вводить привязки в глобальную область видимости или в область видимости интерпретатора сценариев, их можно накапливать в объекте типа `Bindings` и передавать методу `eval()`, как показано ниже. Это очень удобно, если набор привязок не требуется сохранять для последующих вызовов метода `eval()`.

```
Bindings scope = engine.createBindings();
scope.put(b, new JButton());
engine.eval(scriptString, scope);
```

НА ЗАМЕТКУ! Безусловно, может возникнуть потребность иметь и другие области видимости, отличающиеся как от глобальной области видимости, так и от области видимости интерпретатора сценариев. Например, веб-контейнеру могут потребоваться области видимости запросов и сеансов. В подобных случаях разработчикам приходится самостоятельно создавать класс, реализующий интерфейс **ScriptContext**, чтобы управлять набором своих областей видимости. Каждая такая область видимости должна снабжаться целочисленным номером,

а поиск должен выполняться в первую очередь в областях видимости с наименьшим номером. (В стандартной библиотеке доступен только класс **SimpleScriptContext**, но он предусматривает лишь глобальную область видимости, а также область видимости интерпретатора сценариев.)

javax.script.ScriptEngine 6

- **Object eval(String *script*)**
- **Object eval(Reader *reader*)**
- **Object eval(String *script*, Bindings *bindings*)**
- **Object eval(Reader *reader*, Bindings *bindings*)**

  Вычисляют сценарий, предоставляемый в символьной строке или средством чтения с учетом заданных привязок.
- **Object get(String *key*)**
- **void put(String *key*, Object *value*)**

  Получают или размещают привязку в области видимости интерпретатора сценариев.
- **Bindings createBindings()**

  Создает пустой объект типа **Bindings**, пригодный для данного интерпретатора сценариев.

javax.script.ScriptEngineManager 6

- **Object get(String *key*)**
- **void put(String *key*, Object *value*)**

  Получают или размещают привязку в глобальной области видимости.

javax.script.Bindings 6

- **Object get(String *key*)**
- **void put(String *key*, Object *value*)**

  Получают или размещают в области видимости привязку, представляемую данным объектом типа **Bindings**.

8.1.3. Переадресация ввода-вывода

Стандартный ввод-вывод можно переадресовывать в сценарии, вызывая метод `setReader()` или `setWriter()` соответственно в контексте сценария, как показано в приведенном ниже примере кода, где любые данные, выводимые с помощью таких функций JavaScript, как `print()` или `println()`, направляются объекту `writer`.

```
var writer = new StringWriter();
engine.getContext().setWriter(new PrintWriter(writer, true));
```

Методы `setReader()` и `setWriter()` воздействуют только на стандартные источники ввода-вывода данных в интерпретаторе сценариев. Например, при выполнении приведенного ниже кода в сценарии JavaScript перенаправлен будет только первый вывод. Интерпретатору сценариев Nashorn ничего неизвестно о стандартном источнике ввода данных, поэтому вызов метода `setReader()` ничего не даст.

```
println("Hello");
java.lang.System.out.println("World");
```

javax.script.ScriptEngine 6

- **ScriptContext getContext()**
  Получает стандартный контекст сценариев для данного механизма.

javax.script.ScriptContext 6

- **Reader getReader()**
- **void setReader(Reader *reader*)**
- **Writer getWriter()**
- **void setWriter(Writer *writer*)**
- **Writer getErrorWriter()**
- **void setErrorWriter(Writer *writer*)**
  Получают или устанавливают поток чтения для вводимых данных или поток записи для обычных или уведомляющих об ошибках выводимых данных.

8.1.4. Вызов функций и методов из сценариев

При наличии многих интерпретаторов сценариев функция может вызываться на языке сценариев и без вычисления конкретного кода сценария. Это удобно, если пользователям разрешается реализовывать службу на избранном ими языке сценариев.

Те интерпретаторы сценариев, которые предоставляют подобные функциональные возможности, реализуют интерфейс `Invocable`. В частности, этот интерфейс реализуется интерпретатором сценариев Nashorn. Чтобы вызвать функцию из сценария, достаточно обратиться к методу `invokeFunction()`, указав в нем имя и параметры требуемой функции:

```
// определить функцию приветствия в JavaScript
engine.eval("function greet(how, whom)
            { return how + ', ' + whom + '!' }");
// вызвать эту функцию с аргументами "Hello", "World"
result = ((Invocable) engine).invokeFunction("greet",
                              "Hello", "World");
```

Если же язык сценариев является объектно-ориентированным, можно вызвать метод `invokeMethod()` следующим образом:

```
// определить класс Greeter в JavaScript:
engine.eval("function Greeter(how) { this.how = how }");
```

```
engine.eval("Greeter.prototype.welcome = "
          + " function(whom) "
          + "{ return this.how + ', ' + whom + '!' }");

// получить экземпляр:
Object yo = engine.eval("new Greeter('Yo')");

// вызвать метод приветствия для экземпляра:
result = ((Invocable) engine).invokeMethod(yo,
                              "welcome", "World");
```

НА ЗАМЕТКУ! Подробнее об определении классов в JavaScript см. в книге *JavaScript—The Good Parts* Дугласа Крокфорда (Douglas Crockford, издательство O'Reilly, 2008 г.).[1]

НА ЗАМЕТКУ! Даже если механизм сценариев не реализует интерфейс **`Invocable`**, метод все равно можно вызвать не зависящим от конкретного языка образом. Метод **`getMethodCallSyntax()`** из класса **`ScriptEngineFactory`** формирует символьную строку, которую можно затем передать методу **`eval()`**. Но все параметры этого метода должны быть привязаны к именам, в то время как метод **`invokeMethod()`** может вызываться с произвольными значениями параметров.

Можно пойти еще дальше и запросить интерпретатор сценариев реализовать интерфейс Java. В этом случае появится возможность вызывать функции и методы из сценариев, используя синтаксис Java для вызова методов. И хотя это зависит от конкретного интерпретатора сценариев, как правило, для каждого метода из интерфейса достаточно предоставить соответствующую функцию. Рассмотрим в качестве примера следующий интерфейс Java:

```
public interface Greeter
{
  String greet(String whom);
}
```

Если определить глобальную функцию с тем же именем в Nashorn, ее можно вызвать через следующий интерфейс:

```
// определить функцию приветствия в JavaScript:
engine.eval("function welcome(whom)"
          + " { return 'Hello, ' + whom + '!' }");
// получить объект Java и вызвать метод Java:
Greeter g = ((Invocable) engine).getInterface(Greeter.class);
result = g.welcome("World");
```

В объектно-ориентированном языке сценариев доступ к классу из сценария можно получить через соответствующий интерфейс Java. В следующем примере кода демонстрируется, каким образом объект класса `SimpleGreeter` из языка JavaScript вызывается в синтаксисе языка Java:

```
Greeter g = ((Invocable) engine)
            .getInterface(yo, Greeter.class);
result = g.welcome("World");
```

[1] В русском переводе эта книга вышла под названием *JavaScript. Сильные стороны* в издательстве "Питер", СПб. 2012 г.

Таким образом, интерфейс `Invocable` оказывается удобным в том случае, если требуется вызвать код сценария из кода Java, не особенно разбираясь в синтаксисе языка сценариев.

javax.script.Invocable 6

- **`Object invokeFunction(String name, Object... parameters)`**
- **`Object invokeMethod(Object implicitParameter, String name, Object... explicitParameters)`**

  Вызывают функцию или метод с указанным именем, передавая заданные параметры.
- **`<T> T getInterface(Class<T> iface)`**

  Возвращает реализацию указанного интерфейса, методы которого реализуются с помощью функций из механизма сценариев.
- **`<T> T getInterface(Object implicitParameter, Class<T> iface)`**

  Возвращает реализацию указанного интерфейса, методы которого реализуются с помощью методов заданного объекта.

8.1.5. Компиляция сценариев

Некоторые интерпретаторы сценариев способны компилировать код сценария в промежуточную форму для более эффективного выполнения. Такие интерпретаторы реализуют интерфейс `Compilable`. В следующем примере кода демонстрируется компилирование и вычисление кода, содержащегося в файле сценария:

```
var reader = new FileReader("myscript.js");
CompiledScript script = null;
if (engine implements Compilable)
    CompiledScript script =
            ((Compilable) engine).compile(reader);
```

После компиляции сценария можно перейти к его выполнению. В приведенном ниже фрагменте кода демонстрируется выполнение скомпилированного кода сценария, если компиляция прошла успешно, а иначе — исходного сценария, если окажется, что механизм сценариев не поддерживает компиляцию. Безусловно, компилировать сценарий нужно лишь в том случае, если его требуется выполнить повторно.

```
if (script != null)
  script.eval();
else
  engine.eval(reader);
```

javax.script.Compilable 6

- **`CompiledScript compile(String script)`**
- **`CompiledScript compile(Reader reader)`**

  Компилируют сценарий, задаваемый символьной строкой или потоком чтения.

javax.script.CompiledScript 6

- **Object eval()**
- **Object eval(Bindings *bindings*)**

  Вычисляют данный сценарий.

8.1.6. Пример создания сценария для обработки событий в пользовательском интерфейсе

Чтобы продемонстрировать возможности прикладного интерфейса API для сценариев, рассмотрим пример программы, позволяющей пользователям задавать обработчики событий на избранном языке сценариев.

Проанализируйте исходный код, приведенный в листинге 8.1. В этом коде средства создания сценариев вводятся в класс произвольного фрейма. По умолчанию это класс `ButtonFrame` из листинга 8.2, аналогичный по своим функциям программе обработки событий, демонстрировавшейся в первом томе настоящего издания, но с двумя отличиями:

- у каждого компонента имеется свой собственный набор свойств `name`;
- отсутствуют обработчики событий.

Требующиеся обработчики событий определяются в файле свойств, а каждое свойство определяется в следующей форме:

имяКомпонента.имяСобытия = кодСценария

Так, если пользователь выбирает язык сценариев JavaScript, требующиеся обработчики событий предоставляются в файле `js.properties` приведенным ниже образом. В сопутствующем коде имеются также файлы для Groovy и R.

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
blueButton.action=panel.background = java.awt.Color.BLUE
redButton.action=panel.background = java.awt.Color.RED
```

Рассматриваемая здесь программа начинается с загрузки интерпретатора сценариев на языке, указываемом в командной строке. Если же язык не указан, то по умолчанию выбирается JavaScript.

Далее выполняется обработка сценария `init.`*`язык`*, если таковой имеется. И это удобно, поскольку интерпретаторы языков R и Scheme нуждаются в ряде громоздких операций инициализации, которые вряд ли стоит включать в каждый сценарий для обработки событий.

После этого осуществляется рекурсивный обход всех дочерних компонентов и ввод привязок (*имя*, *`объект`*) в область видимости интерпретатора сценариев. Далее выполняется чтение из файла свойств *`язык`*`.properties`. Для каждого свойства конструируется прокси-объект, замещающий обработчик событий, который, собственно, и заставляет выполняться код сценария. Подробности реализации механизма замещения носят несколько технический характер, поэтому тем, кто желает разобраться в нем, рекомендуется еще раз прочитать раздел,

посвященный прокси-объектам в главе 6 первого тома настоящего издания. Но самое главное, что каждый обработчик событий вызывает следующий метод:

```
engine.eval(scriptCode);
```

Остановимся подробнее на обработке событий от кнопки выбора желтого цвета фона (объекте `yellowButton`). При обработке приведенной ниже строки кода обнаруживается компонент `JButton` под именем `"yellowButton"`.

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
```

Далее к этому компоненту присоединяется объект типа `ActionListener` с методом `actionPerformed()`, который выполняет сценарий, если он создан средствами Nashorn:

```
panel.background = java.awt.Color.YELLOW
```

Интерпретатор сценариев содержит привязку, которая связывает имя `"panel"` с объектом типа `JPanel`. Когда наступает событие, выполняется метод `setBackground()` для этого объекта, а в итоге изменяется цвет фона панели.

Запустить рассматриваемую здесь программу с обработчиками событий из сценария JavaScript можно, выполнив команду

```
java ScriptTest
```

А для того чтобы использовать обработчики событий из сценария Groovy, нужно выполнить такую команду:

```
java -classpath .:groovy/lib/\* ScriptTest groovy
```

где *groovy* — каталог, в котором установлен язык Groovy. Для реализации языка R по проекту Renjin архивные JAR-файлы для библиотеки Renjin Studio и интерпретатора сценариев Renjin следует включить в путь к соответствующим классам. Оба эти компонента свободно доступны для загрузки по адресу `www.renjin.org/downloads.html`.

В рассматриваемом здесь примере программы демонстрируется применение сценариев при программировании графического пользовательского интерфейса на платформе Java. Желающие могут пойти еще дальше и описать такой интерфейс с помощью XML-файла, как было показано в главе 3. В этом случае данная программа превратится в интерпретатор графических пользовательских интерфейсов с визуальным представлением, определяемым в формате XML, а также поведением, определяемым на языке сценариев. Это очень похоже на среду создания динамических серверных сценариев и динамических HTML-страниц.

Листинг 8.1. Исходный код из файла **`script/ScriptTest.java`**

```
1  package script;
2
3  import java.awt.*;
4  import java.beans.*;
5  import java.io.*;
6  import java.lang.reflect.*;
7  import java.util.*;
8  import javax.script.*;
9  import javax.swing.*;
```

```
10
11  /**
12   * @version 1.03 2018-05-01
13   * @author Cay Horstmann
14   */
15  public class ScriptTest
16  {
17    public static void main(String[] args)
18    {
19      EventQueue.invokeLater(() ->
20      {
21        try
22        {
23          var manager = new ScriptEngineManager();
24          String language;
25          if (args.length == 0)
26          {
27            System.out.println("Available factories: ");
28            for (ScriptEngineFactory factory :
29                 manager.getEngineFactories())
30              System.out.println(factory.getEngineName());
31
32            language = "nashorn";
33          }
34          else language = args[0];
35
36          final ScriptEngine engine =
37                manager.getEngineByName(language);
38          if (engine == null)
39          {
40            System.err.println("No engine for "
41                               + language);
42            System.exit(1);
43          }
44
45          final String frameClassName = args.length < 2
46                      ? "buttons1.ButtonFrame" : args[1];
47
48          var frame = (JFrame) Class
49                      .forName(frameClassName)
50                      .getConstructor().newInstance();
51          InputStream in = frame.getClass()
52                .getResourceAsStream("init." + language);
53          if (in != null)
54            engine.eval(new InputStreamReader(in));
55          var components =
56              new HashMap<String, Component>();
57          getComponentBindings(frame, components);
58          components.forEach((name, c) ->
59                      engine.put(name, c));
60
61          var events = new Properties();
62          in = frame.getClass().getResourceAsStream(
63                      language + ".properties");
64          events.load(in);
65
```

```
66          for (Object e : events.keySet())
67          {
68            String[] s = ((String) e).split("\\.");
69            addListener(s[0], s[1],
70                        (String) events.get(e),
71                        engine, components);
72          }
73          frame.setTitle("ScriptTest");
74          frame.setDefaultCloseOperation(
75                JFrame.EXIT_ON_CLOSE);
76          frame.setVisible(true);
77        }
78        catch (ReflectiveOperationException
79               | IOException | ScriptException
80               | IntrospectionException ex)
81        {
82          ex.printStackTrace();
83        }
84      });
85    }
86
87    /**
88     * Собирает все именованные компоненты в контейнер
89     * @param c Компонент
90     * @param namedComponents Отображение, в которое
91     *                        вводятся все компоненты
92     *                        и их имена
93     */
94    private static void getComponentBindings(Component c,
95            Map<String, Component> namedComponents)
96    {
97      String name = c.getName();
98      if (name != null) { namedComponents.put(name, c); }
99      if (c instanceof Container)
100     {
101       for (Component child :
102            ((Container) c).getComponents())
103         getComponentBindings(child, namedComponents);
104     }
105   }
106
107   /**
108    * Вводит в объект приемник событий, метод которого
109    * выполняет сценарий
110    * @param beanName Имя компонента JavaBeans,
111    *                 в который вводится приемник событий
112    * @param eventName Имя компонента JavaBeans,
113    *                  например, "action" (действие)
114    *                  или "change" (изменение)
115    * @param scriptCode Выполняемый код сценария
116    * @param engine Интерпретатор, выполняющий
117    *               код сценария
118    * @param bindings Привязки для выполнения сценария
119    * @throws Исключение типа IntrospectionException
120    */
121   private static void addListener(String beanName,
```

```
122           String eventName, final String scriptCode,
123           ScriptEngine engine, Map<String,
124           Component> components)
125           throws ReflectiveOperationException,
126                  IntrospectionException
127   {
128     Object bean = components.get(beanName);
129     EventSetDescriptor descriptor =
130             getEventSetDescriptor(bean, eventName);
131     if (descriptor == null) return;
132     descriptor.getAddListenerMethod()
133               .invoke(bean, Proxy.newProxyInstance(
134                       null, new Class[]
135                       { descriptor.getListenerType() },
136                       (proxy, method, args) ->
137                         {
138                           engine.eval(scriptCode);
139                           return null;
140                         }));
141   }
142
143   private static EventSetDescriptor
144           getEventSetDescriptor(
145                     Object bean, String eventName)
146           throws IntrospectionException
147   {
148     for (EventSetDescriptor descriptor :
149          Introspector.getBeanInfo(bean.getClass())
150                      .getEventSetDescriptors())
151       if (descriptor.getName().equals(eventName))
152         return descriptor;
153     return null;
154   }
155 }
```

Листинг 8.2. Исходный код из файла **buttons1/ButtonFrame.java**

```
1  package buttons1;
2
3  import javax.swing.*;
4
5  /**
6   * Фрейм с панелью кнопок
7   * @version 1.00 2007-11-02
8   * @author Cay Horstmann
9   */
10 public class ButtonFrame extends JFrame
11 {
12   private static final int DEFAULT_WIDTH = 300;
13   private static final int DEFAULT_HEIGHT = 200;
14
15   private JPanel panel;
16   private JButton yellowButton;
17   private JButton blueButton;
18   private JButton redButton;
```

```
19
20    public ButtonFrame()
21    {
22       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
23
24       panel = new JPanel();
25       panel.setName("panel");
26       add(panel);
27
28       yellowButton = new JButton("Yellow");
29       yellowButton.setName("yellowButton");
30       blueButton = new JButton("Blue");
31       blueButton.setName("blueButton");
32       redButton = new JButton("Red");
33       redButton.setName("redButton");
34
35       panel.add(yellowButton);
36       panel.add(blueButton);
37       panel.add(redButton);
38    }
39 }
```

8.2. Прикладной интерфейс API для компилятора

Имеется немало инструментальных средств, в которых требуется вызывать компилятор Java. К их числу, очевидно, относятся среды разработки и средства обучения программированию на Java, а также инструментальные средства, автоматизирующие процессы тестирования и построения прикладного кода. Еще одним тому примером служит обработка веб-страниц типа JSP (JavaServer Pages) со встроенными операторами Java.

8.2.1. Вызов компилятора

Компилятор вызывается очень просто, как показано в приведенном ниже примере кода. Получаемое в итоге нулевое значение переменной `result` указывает на удачный исход компиляции.

```
JavaCompiler compiler =
    ToolProvider.getSystemJavaCompiler();
OutputStream outStream = ...;
OutputStream errStream = ...;
int result = compiler.run(null, outStream, errStream,
                          "-sourcepath", "src", "Test.java");
```

Все выводимые данные и сообщения об ошибках компилятор направляет в указанные потоки вывода. В качестве параметров метода `run()` можно указывать и пустое значение `null`. В данном случае используются стандартные потоки вывода `System.out` и `System.err`. Первый параметр метода `run()` обозначает поток ввода, но, поскольку никаких данных, вводимых с консоли, компилятор не принимает, значение этого параметра всегда оставляется пустым (`null`). Сам же метод `run()` наследуется из обобщенного интерфейса `Tool`, допускающего применение инструментальных средств для чтения вводимых данных.

Остальные параметры метода `run()` являются аргументами, которые следовало бы передать утилите `javac`, если бы этот метод вызывался из командной строки. Они могут обозначать как параметры командной строки, так и имена файлов.

8.2.2. Запуск заданий на компиляцию

С помощью объекта типа `CompilationTask` можно получить еще больший контроль над процессом компиляции. Это может быть удобно в том случае, если требуется предоставить исходный код из символьной строки, зафиксировать файлы классов в оперативной памяти или обработать сообщения или предупреждения об ошибках или неполадках.

Чтобы получить задание на компиляцию в виде объекта типа `CompilationTask`, необходимо получить сначала объект `compiler`, как было показано в предыдущем разделе, а затем сделать следующий вызов:

```
JavaCompiler.CompilationTask task = compiler.getTask(
        // если указано значение null этого параметра,
        // то используется поток вывода System.err:
      errorWriter,
        // если указано значение null этого параметра,
        // то используется стандартный диспетчер файлов:
      fileManager,
        // если указано значение null этого параметра,
        // то используется поток вывода System.err:
      diagnostics,
        // если конкретное значение этого параметра не
        // указано, он принимает пустое значение null:
      options,
        // этот параметр служит для обработки аннотаций;
        // если конкретное значение этого параметра не
        // указано, он принимает пустое значение null:
      classes,
      sources);
```

Три последних параметра в приведенном выше вызове являются экземплярами типа `Iterable`. Например, последовательность параметров компиляции может быть задана следующим образом:

```
Iterable<String> options = List.of("-d", "bin");
```

В качестве параметра `sources` указывается итератор типа `Iterable` экземпляров типа `JavaFileObject`, представляющих исходные файлы. Если требуется скомпилировать файлы, находящиеся на жестком диске, следует получить стандартный диспетчер файлов в виде объекта типа `StandardJavaFileManager` и вызвать его метод `getJavaFileObjects()`:

```
StandardJavaFileManager fileManager =
       compiler.getStandardFileManager(null, null, null);
Iterable<JavaFileObject> sources =
        fileManager.getJavaFileObjectsFromStrings(
                    List.of("File1.java", "File2.java"));
JavaCompiler.CompilationTask task = compiler.getTask(
            null, null, null, options, null, sources);
```

НА ЗАМЕТКУ! Параметр **classes** служит лишь для обработки аннотаций. И в этом случае необходимо также сделать вызов **task.processors(annotationProcessors)** со списком объектов типа **Processor**. Характерный пример обработки аннотаций приведен в разделе 8.6.

Метод `getTask()` возвращает объект задания, но пока еще не запускает процесс компиляции. Класс `CompilationTask` реализует интерфейс `Callable<Boolean>`. Объект этого класса можно передать исполнителю типа `ExecutorService` для параллельного исполнения или же сделать синхронный вызов, как показано ниже.

```
Boolean success = task.call();
```

8.2.3. Фиксация диагностики

Для приема появляющихся сообщений об ошибках устанавливается приемник диагностики, реализующий интерфейс `DiagnosticListener`. Всякий раз, когда компилятор выдает предупреждение или сообщение об ошибке, этот приемник получает объект типа `Diagnostic`. В частности, интерфейс `DiagnosticListener` реализуется в классе `DiagnosticCollector`, где собираются все диагностические данные для просмотра и анализа по завершении компиляции, как демонстрируется в следующем примере кода:

```
DiagnosticCollector<JavaFileObject> collector =
          new DiagnosticCollector<>();
compiler.getTask(null, fileManager, collector, null,
                 null, sources).call();
for (Diagnostic<? extends JavaFileObject> d :
     collector.getDiagnostics())
{
  System.out.println(d);
}
```

Объект типа `Diagnostic` содержит сведения о месте появления ошибки компиляции, включая имя файла, номер строки и столбца, а также удобочитаемое описание ошибки.

Если требуется перехватывать сообщения об отсутствующих файлах, приемник диагностики типа `DiagnosticListener` можно установить в стандартном диспетчере файлов:

```
StandardJavaFileManager fileManager =
  compiler.getStandardFileManager(diagnostics, null, null);
```

8.2.4. Чтение исходных файлов из оперативной памяти

Чтобы оперативно сгенерировать исходный код, его можно скомпилировать из оперативной памяти, не сохраняя исходные файлы на жестком диске. В частности, для хранения скомпилированного кода можно воспользоваться следующим классом:

```
public class StringSource extends SimpleJavaFileObject
{
  private String code;
```

```
  StringSource(String name, String code)
  {
    super(URI.create("string:///" + name.replace('.','/')
                  + ".java"), Kind.SOURCE);
    this.code = code;
  }

  public CharSequence
       getCharContent(boolean ignoreEncodingErrors)
  {
    return code;
  }
}
```

Далее остается лишь сгенерировать код отдельных классов и предоставить компилятору список объектов типа `StringSource`:

```
List<StringSource> sources = List.of(new StringSource(
                  className1, class1CodeString), . . .);
task = compiler.getTask(null, fileManager, diagnostics,
                        null, null, sources);
```

8.2.5. Запись байт-кодов в оперативную память

Если исходный код классов компилируется оперативно, то сохранять файлы классов на жестком диске нет необходимости. Их можно сохранить в оперативной памяти, чтобы сразу же загрузить их оттуда. Для этого прежде всего создается следующий класс, в котором хранятся получаемые в итоге байт-коды:

```
public class ByteArrayClass extends SimpleJavaFileObject
{
  private ByteArrayOutputStream out;

  ByteArrayClass(String name)
  {
    super(URI.create("bytes:///" + name.replace('.','/')
                  + ".class"), Kind.CLASS);
  }

  public byte[] getCode()
  {
    return out.toByteArray();
  }

  public OutputStream openOutputStream() throws IOException
  {
    out = new ByteArrayOutputStream();
    return out;
  }
}
```

Затем необходимо сконфигурировать диспетчер файлов, в которые требуется выводить скомпилированные классы.

```
List<ByteArrayClass> classes = new ArrayList<>();
StandardJavaFileManager stdFileManager =
        compiler.getStandardFileManager(null, null, null);
```

```
JavaFileManager fileManager =
    new ForwardingJavaFileManager<JavaFileManager>
        (stdFileManager)
{
  public JavaFileObject getJavaFileForOutput(
        Location location, String className,
        Kind kind, FileObject sibling)
        throws IOException
  {
    if (kind == Kind.CLASS)
    {
      ByteArrayClass outfile =
              new ByteArrayClass(className);
      classes.add(outfile);
      return outfile;
    }
    else
      return super.getJavaFileForOutput(
                  location, className, kind, sibling);
  }
};
```

Далее для загрузки классов потребуется соответствующий загрузчик (см. главу 10):

```
public class ByteArrayClassLoader extends ClassLoader
{
  private Iterable<ByteArrayClass> classes;

  public ByteArrayClassLoader(
        Iterable<ByteArrayClass> classes)
  {
    this.classes = classes;
  }

  public Class<?> findClass(String name)
        throws ClassNotFoundException
  {
    for (ByteArrayClass cl : classes)
    {
      if (cl.getName().equals("/" + name.replace('.','/')
                          + ".class"))
      {
        byte[] bytes = cl.getCode();
        return defineClass(name, bytes, 0, bytes.length);
      }
    }
    throw new ClassNotFoundException(name);
  }
}
```

По окончании компиляции следует вызвать метод `Class.forName()` с данным загрузчиком классов:

```
ByteArrayClassLoader loader =
        new ByteArrayClassLoader(classes);
Class<?> cl = Class.forName(className, true, loader);
```

8.2.6. Пример динамического генерирования кода Java

В технологии JSP для формирования веб-страниц в динамическом режиме допускается сочетать HTML-разметку с фрагментами исходного кода Java, как демонстрируется в следующем примере:

```
<p>The current date and time is
  <b><%= new java.util.Date() %></b>.</p>
```

Интерпретатор JSP динамически компилирует код Java в сервлет. В рассматриваемом здесь примере прикладной программы демонстрируется более простой пример динамического генерирования кода Swing. Основной замысел состоит в том, чтобы воспользоваться построителем графического пользовательского интерфейса с целью расположить компоненты во фрейме и обозначить их поведение во внешнем файле. В листинге 8.4 приведен очень простой пример класса фрейма, а в листинге 8.5 — исходный код для обозначения действий экранных кнопок. Следует заметить, что конструктор класса фрейма вызывает абстрактный метод `addEventHandlers()`, а генератор кода создает подкласс, реализующий метод `addEventHandlers()`, вводя приемник для обработки действий в каждой строке из файла свойств `action.properties`. (Расширение возможностей генерировать код для обработки других типов событий оставляем читателю в качестве упражнения для самостоятельной проработки.)

Создаваемый в итоге подкласс размещается в пакете под именем `x`, которое нигде больше не должно использоваться в данной программе. Сгенерированный код принимает следующую форму:

```
package x;
public class Frame extends ИмяСуперкласса
{
  protected void addEventHandlers()
  {
     ИмяКомпонента1.addActionListener(event ->
       {
         код первого обработчика событий
       });
    // повторить для остальных обработчиков событий . . .
  }
}
```

Метод `buildSource()` в программе из листинга 8.3 служит для генерирования этого кода и его размещения в объекте типа `StringBuilderJavaSource`. Затем этот объект передается компилятору Java.

Как пояснялось в предыдущем разделе, построение объектов типа `ByteArrayJavaClass` для каждого класса из пакета `x` осуществляется методом `getJavaFileForOutput()` из класса `ForwardingJavaFileManager`. Эти объекты перехватывают файлы классов, формируемые при компиляции класса `x.Frame`. Метод `getJavaFileForOutput()` вводит каждый объект файла в список прежде, чем возвратить его, чтобы в дальнейшем можно было обнаружить байт-коды.

После компиляции классы, которые хранятся в данном списке, загружаются с помощью загрузчика классов, упоминавшегося в предыдущем разделе. Затем конструируется объект класса для отображения фрейма рассматриваемой здесь прикладной программы:

```
var loader = new ByteArrayClassLoader(classFileObjects);
var frame = (JFrame) loader.loadClass("x.Frame")
                           .getConstructor().newInstance();
frame.setVisible(true);
```

Если щелкнуть на экранных кнопках, цвет фона изменится обычным образом. А для того чтобы убедиться, что действия экранных кнопок в самом деле компилируются динамически, измените какую-нибудь из строк в файле свойств `action.properties`, например, следующим образом:

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW);
yellowButton.setEnabled(false);
```

После этого запустите еще раз данную программу на выполнение. Если теперь щелкнуть на кнопке Yellow, она станет недоступной. Загляните также в каталоги с исходным кодом. Вы не обнаружите там ни исходных файлов, ни файлов классов из пакета `x`. Данный пример наглядно демонстрирует применение динамической компиляции вместе с сохранением файлов исходного кода и классов в оперативной памяти.

Листинг 8.3. Исходный код из файла **`compiler/CompilerTest.java`**

```
1   package compiler;
2
3   import java.awt.*;
4   import java.io.*;
5   import java.nio.file.*;
6   import java.util.*;
7   import java.util.List;
8
9   import javax.swing.*;
10  import javax.tools.*;
11  import javax.tools.JavaFileObject.*;
12
13  /**
14   * @version 1.10 2018-05-01
15   * @author Cay Horstmann
16   */
17  public class CompilerTest
18  {
19    public static void main(final String[] args)
20         throws IOException, ReflectiveOperationException
21    {
22      JavaCompiler compiler =
23                  ToolProvider.getSystemJavaCompiler();
24
25      var classFileObjects =
26                new ArrayList<ByteArrayClass>();
27      var diagnostics =
28          new DiagnosticCollector<JavaFileObject>();
29
30      JavaFileManager fileManager =
31              compiler.getStandardFileManager(
32                      diagnostics, null, null);
33      fileManager = new ForwardingJavaFileManager
34                       <JavaFileManager>(fileManager)
```

```
35        {
36           public JavaFileObject getJavaFileForOutput(
37                 Location location, String className,
38                 Kind kind, FileObject sibling)
39                 throws IOException
40           {
41             if (kind == Kind.CLASS)
42             {
43               var fileObject =
44                   new ByteArrayClass(className);
45               classFileObjects.add(fileObject);
46               return fileObject;
47             }
48             else return super.getJavaFileForOutput(
49                 location, className, kind, sibling);
50           }
51        };
52
53        String frameClassName = args.length == 0
54                   ? "buttons2.ButtonFrame" : args[0];
55        // compiler.run(null, null, null,
56        //     frameClassName.replace(".", "/") + ".java");
57
58        StandardJavaFileManager fileManager2 = compiler
59                .getStandardFileManager(null, null, null);
60        var sources = new ArrayList<JavaFileObject>();
61        for (JavaFileObject o :
62             fileManager2.getJavaFileObjectsFromStrings(
63                          List.of(frameClassName.replace(
64                                  ".", "/") + ".java")))
65          sources.add(o);
66
67        JavaFileObject source = buildSource(frameClassName);
68        JavaCompiler.CompilationTask task = compiler
69                   .getTask(null, fileManager, diagnostics,
70                            null, null, List.of(source));
71        Boolean result = task.call();
72
73        for (Diagnostic<? extends JavaFileObject> d :
74             diagnostics.getDiagnostics())
75          System.out.println(d.getKind() + ": "
76                           + d.getMessage(null));
77        fileManager.close();
78        if (!result)
79        {
80          System.out.println("Compilation failed.");
81          System.exit(1);
82        }
83
84        var loader =
85            new ByteArrayClassLoader(classFileObjects);
86        var frame = (JFrame) loader.loadClass("x.Frame")
87                   .getConstructor().newInstance();
88
89        EventQueue.invokeLater(() ->
90          {
91            frame.setDefaultCloseOperation(
```

```
92                          JFrame.EXIT_ON_CLOSE);
93          frame.setTitle("CompilerTest");
94          frame.setVisible(true);
95        });
96    }
97
98    /*
99     * Генерирует исходный код подкласса, реализующего
100    * метод addEventHandlers()
101    * @return Объект файла, содержащий исходный код в
102    *         построителе символьных строк
103    */
104   static JavaFileObject buildSource(
105         String superclassName)
106         throws IOException, ClassNotFoundException
107   {
108     var builder = new StringBuilder();
109     builder.append("package x;\n\n");
110     builder.append("public class Frame extends "
111                 + superclassName + " {\n");
112     builder.append(
113             "protected void addEventHandlers() {\n");
114     var props = new Properties();
115     props.load(Files.newInputStream(Paths.get(
116        superclassName.replace(".", "/")).getParent()
117                     .resolve("action.properties")));
118     for (Map.Entry<Object, Object> e : props.entrySet())
119     {
120       var beanName = (String) e.getKey();
121       var eventCode = (String) e.getValue();
122       builder.append(beanName
123              + ".addActionListener(event -> {\n");
124       builder.append(eventCode);
125       builder.append("\n} );\n");
126     }
127     builder.append("} }\n");
128     return new StringSource("x.Frame", builder.toString());
129   }
130 }
```

Листинг 8.4. Исходный код из файла **buttons2/ButtonFrame.java**

```
1  package buttons2;
2  import javax.swing.*;
3
4  /**
5   * Фрейм с панелью кнопок
6   * @version 1.00 2007-11-02
7   * @author Cay Horstmann
8  */
9  public abstract class ButtonFrame extends JFrame
10 {
11   public static final int DEFAULT_WIDTH = 300;
12   public static final int DEFAULT_HEIGHT = 200;
13
```

```
14   protected JPanel panel;
15   protected JButton yellowButton;
16   protected JButton blueButton;
17   protected JButton redButton;
18
19   protected abstract void addEventHandlers();
20
21   public ButtonFrame()
22   {
23     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
24
25     panel = new JPanel();
26     add(panel);
27
28     yellowButton = new JButton("Yellow");
29     blueButton = new JButton("Blue");
30     redButton = new JButton("Red");
31
32     panel.add(yellowButton);
33     panel.add(blueButton);
34     panel.add(redButton);
35
36     addEventHandlers();
37   }
38 }
```

Листинг 8.5. Исходный код из файла **buttons2/action.properties**

```
1 yellowButton=panel.setBackground(java.awt.Color.YELLOW);
2 blueButton=panel.setBackground(java.awt.Color.BLUE);
```

javax.tools.Tool 6

- **int run(InputStream *in*, OutputStream *out*, OutputStream *err*, String... *arguments*)**

  Запускает утилиту компиляции с указанными потоками ввода и вывода, а также потоком вывода сообщений об ошибках и аргументами командной строки. При удачном исходе компиляции возвращает нулевое значение, при неудачном — ненулевое.

javax.tools.JavaCompiler 6

- **StandardJavaFileManager getStandardFileManager(DiagnosticListener<? super JavaFileObject> *diagnosticListener*, Locale *locale*, Charset *charset*)**

  Получает стандартный диспетчер файлов для данного компилятора. Имеется возможность использовать стандартные сообщения об ошибках, региональные настройки и набор символов, указав в качестве соответствующих параметров пустые значения **null**.

javax.tools.JavaCompiler 6 *(окончание)*

- **JavaCompiler.CompilationTask getTask(Writer *out*, JavaFileManager *fileManager*, DiagnosticListener<? super JavaFileObject> *diagnosticListener*, Iterable<String> *options*, Iterable<String> *classesForAnnotationProcessing*, Iterable<? extends JavaFileObject> *sourceFiles*)**
  Получает задание на компиляцию, при вызове которого будут компилироваться указанные исходные файлы. Подробнее об этом см. в предыдущем разделе.

javax.tools.StandardJavaFileManager 6

- **Iterable<? extends JavaFileObject> getJavaFileObjectsFromStrings(Iterable<String> *fileNames*)**
- **Iterable<? extends JavaFileObject> getJavaFileObjectsFromFiles(Iterable<? extends File> *files*)**
  Преобразуют последовательность файлов или их имен в последовательность экземпляров типа **JavaFileObject**.

javax.tools.JavaCompiler.CompilationTask 6

- **Boolean call()**
  Выполняет задание на компиляцию.

javax.tools.DiagnosticCollector<S> 6

- **DiagnosticCollector()**
  Создает пустой сборщик данных.
- **List<Diagnostic<? extends S>> getDiagnostics()**
  Получает собранные диагностические данные.

javax.tools.Diagnostic<S> 6

- **S getSource()**
  Получает исходный объект, связанный с данной процедурой диагностики.
- **Diagnostic.Kind getKind()**
  Получает тип данной процедуры диагностики, принимающий значение одной из следующих констант: **ERROR**, **WARNING**, **MANDATORY_WARNING**, **NOTE** или **OTHER**.
- **String getMessage(Locale *locale*)**
  Получает сообщение, описывающее ошибку, обнаруженную в данной процедуре диагностики. Имеется возможность использовать стандартные региональные настройки, передав пустое значение **null** в качестве соответствующего параметра.

javax.tools.Diagnostic<S> 6 *(окончание)*

- **long getLineNumber()**
- **long getColumnNumber()**

  Получают местоположение ошибки, обнаруженной в данной процедуре диагностики

javax.tools.SimpleJavaFileObject 6

- **CharSequence getCharContent(boolean *ignoreEncodingErrors*)**

  Этот метод переопределяется для объекта файла, представляющего исходный файл и генерирующего исходный код.
- **OutputStream openOutputStream()**

  Этот метод переопределяется для объекта файла, представляющего файл класса и формирующего поток вывода, в который можно направлять генерируемые байт-коды.

javax.tools.ForwardingJavaFileManager<M extends JavaFileManager> 6

- **protected ForwardingJavaFileManager(M *fileManager*)**

  Создает объект типа **JavaFileManager**, делегирующий все вызовы указанному диспетчеру файлов.
- **FileObject getFileForOutput(JavaFileManager.Location *location*, String *className*, JavaFileObject.Kind *kind*, FileObject *sibling*)**

  Вызов этого метода перехватывается, если требуется заменить объект файла для записи файлов классов. Параметр ***kind*** может принимать значение одной из следующих констант: **SOURCE**, **CLASS**, **HTML** или **OTHER**.

8.3. Применение аннотаций

Аннотациями называются дескрипторы, которые разработчики вставляют в свой исходный код, чтобы их можно было обработать соответствующими инструментальными средствами. Эти инструментальные средства могут действовать как на уровне исходного кода, так и на уровне файлов классов, в которых компилятор размещает аннотации. Аннотации не влияют на способ компиляции программ. Компилятор Java генерирует одинаковые инструкции виртуальной машины как с аннотациями, так и без них.

Чтобы извлечь наибольшую пользу из аннотаций, необходимо выбрать подходящее *средство обработки*. В исходный код следует вводить такие аннотации, которые распознаются избранным средством обработки, способным правильно интерпретировать их и выполнять соответствующие действия над исходным кодом. У аннотаций существует немало областей применения, поэтому их универсальность может поначалу вызывать недоразумения. Ниже перечислены некоторые из областей применения аннотаций.

- Автоматическое генерирование вспомогательных файлов, например, файлов дескрипторов развертывания или классов информации о компонентах JavaBeans.
- Автоматическое генерирование кода для тестирования, протоколирования, семантической обработки транзакций и т.д.

8.3.1. Введение в аннотации

Начнем обсуждение аннотаций с основных понятий и продемонстрируем их практическое применение на конкретном примере, пометив методы как приемники событий для компонентов AWT и представив обработчик аннотаций, способный анализировать аннотации и подключать приемники событий. Далее мы подробно рассмотрим синтаксические правила. Наконец, будут продемонстрированы два расширенных примера обработки аннотаций: первый — на уровне исходного кода, второй — на уровне файлов классов, где библиотека Apache Bytecode Engineering Library применяется для вставки дополнительных байт-кодов в аннотированные методы.

Ниже приведен пример объявления простой аннотации. В частности, аннотация `@Test` служит для аннотирования метода `checkRandomInsertions()`.

```
public class MyClass
{
  . . .
  @Test public void checkRandomInsertions()
}
```

Аннотация применяется в Java подобно модификатору и размещается перед аннотируемым элементом *без* точки с запятой. (*Модификатор* — это ключевое слово вроде `public` или `static`.) Имя каждой аннотации предваряется знаком `@` подобно тому, как это делается в документирующих комментариях, автоматически составляемых в формате Javadoc. Но документирующие комментарии в формате Javadoc размещаются между разделителями `/**...*/`, тогда как аннотации являются частью исходного кода.

Сама аннотация `@Test` ничего не делает. Чтобы она смогла приносить какую-то пользу, ей потребуется подходящее инструментальное средство. Например, инструментальное средство модульного тестирования JUnit 5 (доступное по адресу `https://junit.org/junit5/`) способно вызвать все помеченные аннотацией `@Test` методы при тестировании класса. Другое инструментальное средство может удалить все тестовые методы из файла класса, чтобы исключить их из исходного кода программы после ее тестирования. Аннотации могут быть определены вместе со своими *элементами*, как демонстрируется в приведенном ниже примере кода.

```
@Test(timeout="10000")
```

Эти элементы могут обрабатываться инструментальными средствами, способными обрабатывать аннотации. Элементы могут выглядеть и по-другому. Подробнее о них речь пойдет далее в этой главе. Помимо методов, аннотациями могут снабжаться классы, поля и локальные переменные, а сами аннотации — размещаться на тех же местах, где и модификаторы типа `public` или `static`.

Как будет показано в разделе 8.4, аннотациями можно также снабжать пакеты, переменные параметров, параметры типа и примеры применения типов данных.

Каждая аннотация должна определяться с помощью *интерфейса аннотаций*. Методы такого интерфейса должны соответствовать элементам определяемой им аннотации. Например, аннотация `@Test` для модульного тестирования средствами JUnit определяется с помощью следующего интерфейса:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test
{
  long timeout() default 0L;
  . . .
}
```

В объявлении `@interface` создается конкретный интерфейс Java, а инструментальные средства, обрабатывающие аннотации, получают объекты, классы которых реализуют сам интерфейс аннотаций. Например, для извлечения элемента `timeout` из конкретной аннотации `Test` инструментальное средство вызывает метод `timeout()`.

Аннотации `@Target` и `@Retention` являются *мета-аннотациями*. Они помечают аннотацию `@Test`, превращая ее в такую аннотацию, которая может применяться только к методам и должна сохраняться при загрузке файла класса в виртуальную машину. Более подробно они рассматриваются в разделе 8.5.3.

В этом разделе были представлены основные понятия метаданных и аннотаций к программам. В следующем разделе будет рассмотрен конкретный пример обработки аннотаций.

НА ЗАМЕТКУ! Убедительные примеры применения аннотаций можно найти в библиотеках **JCommander** (**http://jcommander.org**) и **picocli** (**https://picocli.info**). Аннотации применяются в этих библиотеках для обработки параметров командной строки.

8.3.2. Пример аннотирования обработчиков событий

Одной из наиболее утомительных задач в программировании пользовательских интерфейсов является присоединение приемников к источникам событий. Многие приемники событий имеют следующий вид:

```
myButton.addActionListener(() -> doSomething());
```

Во избежание подобных хлопот в этом разделе будет разработана специальная аннотация, определение которой приведено далее в листинге 8.8, а ее общая форма представлена ниже.

```
@ActionListenerFor(source="myButton")
       void doSomething() { . . . }
```

Разработчику больше не придется делать вызовы метода `addActionListener()`. Вместо этого каждый метод приемника событий снабжается соответствующей аннотацией. В листинге 8.7 представлен класс `ButtonFrame`, рассматривавшийся в главе 10 первого тома настоящего издания и реализованный заново вместе с подобными аннотациями. Для этого придется также реализовать интерфейс аннотаций, как демонстрируется в листинге 8.8.

Разумеется, сами аннотации ничего не делают — они хранятся в исходном файле. Компилятор размещает их в файле классов, а виртуальная машина загружает их. Следовательно, требуется какой-то механизм для анализа аннотаций и установки приемников действий. Эта обязанность возлагается на класс `ActionListenerInstaller`. В конструкторе класса `ButtonFrame` вызывается следующий метод из класса `ActionListenerInstaller`:

```
ActionListenerInstaller.processAnnotations(this);
```

В статическом методе `processAnnotations()` перечисляются все методы объекта, которые он получает. Для каждого метода извлекается и обрабатывается объект аннотации типа `ActionListenerFor`, как показано ниже.

```
Class<?> cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
  ActionListenerFor a =
        m.getAnnotation(ActionListenerFor.class);
  if (a != null) . . .
}
```

В данном случае применяется метод `getAnnotation()`, определенный в интерфейсе `AnnotatedElement`, который реализуют такие классы, как `Method`, `Constructor`, `Field`, `Class` и `Package`. Имя исходного поля хранится в объекте аннотации. Оно извлекается в результате вызова метода `source()`, а затем осуществляется поиск соответствующего ему поля, как следует из приведенного ниже примера кода.

```
String fieldName = a.source();
Field f = cl.getDeclaredField(fieldName);
```

Этот пример наглядно показывает ограниченность рассматриваемой здесь аннотации. Исходный элемент должен обозначать имя поля, а представлять локальную переменную он не может.

Остальная часть кода из рассматриваемого здесь примера программы носит довольно технический характер. В частности, для каждого аннотируемого метода создается прокси-объект, класс которого реализует интерфейс `ActionListener` с методом `actionPerformed()`, вызывающим данный аннотируемый метод. (Подробнее о прокси-объектах см. в главе 6 первого тома настоящего издания.) Не особенно вдаваясь в подробности, которые здесь не так важны, отметим самое главное: функциональные возможности аннотаций устанавливаются с помощью метода `processAnnotations()`. Весь процесс обработки аннотаций схематически представлен на рис. 8.1.

В рассматриваемом здесь примере программы аннотации обрабатываются во время выполнения. Но их можно обрабатывать и на уровне исходного кода. В частности, генератор исходного кода может сгенерировать код для ввода приемников событий. С другой стороны, аннотации можно также обрабатывать на уровне байт-кода. Для этого редактор байт-кода может вставлять вызовы метода `addActionListener()` в конструктор фрейма. На первый взгляд эта задача кажется слишком сложной, но в настоящее время доступны библиотеки, упрощающие ее решение, как будет показано на конкретном примере, представленном в разделе 8.7.

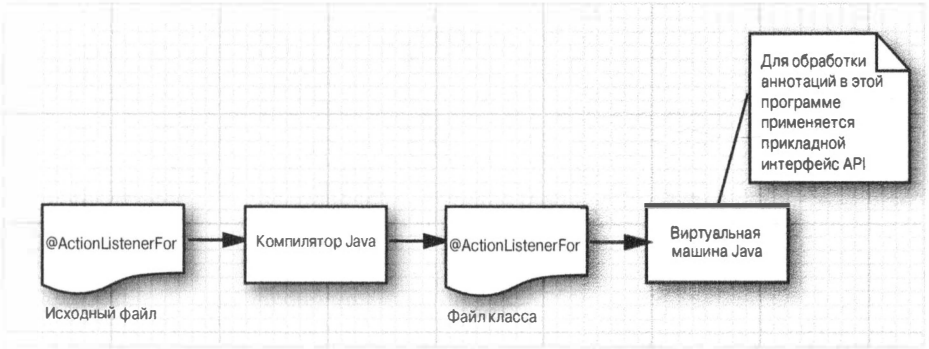


Рис. 8.1. Обработка аннотаций во время выполнения

Пример, рассматриваемый в этом разделе, ни в коей мере нельзя рекомендовать в качестве серьезного средства для разработки пользовательских интерфейсов. Применение служебного метода для ввода приемников событий может оказаться не менее удобным, чем вставка аннотаций в исходный код. (На самом деле попытка реализовать именно такой подход предпринята в классе `java.beans.EventHandler`. Этот класс нетрудно сделать действительно полезным, предоставив метод, вводящий обработчик событий, вместо того чтобы создавать его.)

Тем не менее данный пример призван продемонстрировать внутренний механизм аннотирования программы и анализа аннотаций. Его рассмотрение должно помочь вам как следует подготовиться к изучению материала последующих разделов, в которых подробно описан синтаксис аннотаций.

Листинг 8.6. Исходный код из файла **`runtimeAnnotations/ActionListenerInstaller.java`**

```
1  package runtimeAnnotations;
2
3  import java.awt.event.*;
4  import java.lang.reflect.*;
5
6  /**
7   * @version 1.00 2004-08-17
8   * @author Cay Horstmann
9  */
10 public class ActionListenerInstaller
11 {
12   /**
13    * Обрабатывает все аннотации типа ActionListenerFor
14    * в заданном объекте
15    * @param obj Объект, методы которого могут иметь
16    *            аннотации типа ActionListenerFor
17    */
18   public static void processAnnotations(Object obj)
19   {
20     try
21     {
```

```
      Class<?> cl = obj.getClass();
      for (Method m : cl.getDeclaredMethods())
      {
        ActionListenerFor a =
              m.getAnnotation(ActionListenerFor.class);
        if (a != null)
        {
          Field f = cl.getDeclaredField(a.source());
          f.setAccessible(true);
          addListener(f.get(obj), obj, m);
        }
      }
    }
    catch (ReflectiveOperationException e)
    {
      e.printStackTrace();
    }
  }

  /**
   * Вводит приемник действий, вызывающий заданный метод
   * @param source Источник событий, в который вводится
   *               приемник действий
   * @param param Неявный параметр метода, вызываемого
                  приемником действий
   * @param m Метод, вызываемый приемником действий
  */
  public static void addListener(Object source,
                final Object param, final Method m)
         throws ReflectiveOperationException
  {
    var handler = new InvocationHandler()
      {
        public Object invoke(Object proxy, Method mm, Object[] args)
               throws Throwable
        {
          return m.invoke(param);
        }
      };

    Object listener = Proxy.newProxyInstance(null,
           new Class[]
               { java.awt.event.ActionListener.class }, handler);
    Method adder = source.getClass().getMethod(
           "addActionListener", ActionListener.class);
    adder.invoke(source, listener);
  }
}
```

Листинг 8.7. Исходный код из файла **buttons3/ButtonFrame.java**

```
1  package buttons3;
2
3  import java.awt.*;
```

```
4  import javax.swing.*;
5  import runtimeAnnotations.*;
6
7  /**
8   * Фрейм с панелью кнопок
9   * @version 1.00 2004-08-17
10  * @author Cay Horstmann
11 */
12 public class ButtonFrame extends JFrame
13 {
14   private static final int DEFAULT_WIDTH = 300;
15   private static final int DEFAULT_HEIGHT = 200;
16
17   private JPanel panel;
18   private JButton yellowButton;
19   private JButton blueButton;
20   private JButton redButton;
21
22   public ButtonFrame()
23   {
24     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
25
26     panel = new JPanel();
27     add(panel);
28     yellowButton = new JButton("Yellow");
29     blueButton = new JButton("Blue");
30     redButton = new JButton("Red");
31
32     panel.add(yellowButton);
33     panel.add(blueButton);
34     panel.add(redButton);
35
36     ActionListenerInstaller.processAnnotations(this);
37   }
38
39   @ActionListenerFor(source = "yellowButton")
40   public void yellowBackground()
41   {
42     panel.setBackground(Color.YELLOW);
43   }
44
45   @ActionListenerFor(source = "blueButton")
46   public void blueBackground()
47   {
48     panel.setBackground(Color.BLUE);
49   }
50
51   @ActionListenerFor(source = "redButton")
52   public void redBackground()
53   {
54     panel.setBackground(Color.RED);
55   }
56 }
```

Листинг 8.8. Исходный код из файла **runtimeAnnotations/ActionListenerFor.java**

```
1  package runtimeAnnotations;
2
3  import java.lang.annotation.*;
4
5  /**
6   * @version 1.00 2004-08-17
7   * @author Cay Horstmann
8  */
9  @Target(ElementType.METHOD)
10 @Retention(RetentionPolicy.RUNTIME)
11 public @interface ActionListenerFor
12 {
13   String source();
14 }
```

java.lang.reflect.AnnotatedElement 5.0

- **boolean isAnnotationPresent(Class<? extends Annotation> *annotationType*)**

  Возвращает логическое значение **true**, если данный элемент снабжен аннотацией указанного типа.

- **<T extends Annotation> T getAnnotation(Class<T> *annotationType*)**

  Получает аннотацию заданного типа или пустое значение **null**, если такая аннотация отсутствует у данного элемента.

- **<T extends Annotation> T[] getAnnotationsByType(Class<T> *annotationType*)** 8

  Получает все аннотации повторяющегося типа (см. раздел 8.5.3) или возвращает массив нулевой длины.

- **Annotation[] getAnnotations()**

  Получает все аннотации, доступные для данного элемента, включая унаследованные. Если доступные аннотации отсутствуют, возвращается массив нулевой длины.

- **Annotation[] getDeclaredAnnotations()**

  Получает все аннотации, объявленные для данного элемента, исключая унаследованные. Если доступные аннотации отсутствуют, возвращается массив нулевой длины.

8.4. Синтаксис аннотаций

В последующих разделах поясняется все, что следует знать о синтаксисе аннотаций.

8.4.1. Интерфейсы аннотаций

Аннотация определяется с помощью своего интерфейса следующим образом:

```
модификаторы @interface ИмяАннотации
{
```

```
  объявлениеЭлемента1
  объявлениеЭлемента2
  . . .
}
```

Каждый элемент может быть объявлен в одной из следующих форм:

```
тип имяЭлемента();
```

или

```
тип имяЭлемента() default значение;
```

Например, приведенная ниже аннотация состоит из двух элементов: `assignedTo` и `severity`.

```
public @interface BugReport
{
  String assignedTo() default "[none]";
  int severity();
}
```

Все интерфейсы аннотаций неявно расширяют интерфейс `java.lang.annotation.Annotation`. Это обычный интерфейс, а *не* интерфейс аннотаций. Предоставляемые им методы можно найти в конце этого раздела, где описывается соответствующий прикладной интерфейс API. Самостоятельно расширять интерфейсы аннотаций нельзя. Иными словами, все интерфейсы аннотаций могут непосредственно расширять только интерфейс `java.lang.annotation.Annotation`. Нельзя также предоставить классы, реализующие интерфейсы аннотаций.

Методы из интерфейса аннотаций могут не иметь ни параметров, ни операторов `throws`. Они не могут быть статическими, обобщенными с параметрами типа или методами с реализацией по умолчанию.

Элемент аннотации может принимать один из следующих типов.

- Примитивный тип (`int`, `short`, `long`, `byte`, `char`, `double`, `float` или `boolean`).
- Строковый тип `String`.
- Тип `Class` (с каким-нибудь необязательным параметром вроде `Class<? extends MyClass>`).
- Перечислимый тип `enum`.
- Тип аннотации.
- Массив перечисленных выше типов (массив массивов не является допустимым типом элемента аннотации).

```
public @interface BugReport
{
  enum Status { UNCONFIRMED, CONFIRMED, FIXED, NOTABUG };
  boolean showStopper() default false;
  String assignedTo() default "[none]";
  Class<?> testCase() default Void.class;
  Status status() default Status.UNCONFIRMED;
  Reference ref() default @Reference(); // тип аннотации
  String[] reportedBy();
}
```

java.lang.annotation.Annotation 5.0

- **Class<? extends Annotation> annotationType()**

  Возвращает объект типа **Class**, который представляет интерфейс аннотаций для данного объекта аннотации. Однако вызов метода **getClass()** для объекта аннотации привел бы к возврату конкретного класса, а не интерфейса.
- **boolean equals(Object *other*)**

  Возвращает логическое значение **true**, если параметр ***other*** обозначает объект, класс которого реализует тот же интерфейс аннотаций, что и данный объект аннотации, и если все элементы этого объекта равны параметру ***other***.
- **int hashCode()**

  Возвращает хеш-код, совместимый с методом **equals()** и получаемый из имени интерфейса аннотаций и значений его элементов.
- **String toString()**

  Возвращает строковое представление, содержащее название интерфейса аннотаций и значения элементов, например **@BugReport(assignedTo=[none], severity=0)**.

8.4.2. Объявление аннотаций

Каждая аннотация имеет следующую форму объявления:

```
@ИмяАннотации(имяЭлемента1=значение1,
              имяЭлемента2=значение2, . . .)
```

В качестве примера ниже приведено объявление аннотации для сообщений о программных ошибках.

```
@BugReport(assignedTo="Harry", severity=10)
```

Порядок следования элементов в аннотации особого значения не имеет. Так, следующая аннотация идентична приведенной выше:

```
@BugReport(severity=10, assignedTo="Harry")
```

Значение, устанавливаемое в объявлении аннотации по умолчанию, используется в том случае, если для элемента аннотации не задано конкретное значение. Ниже приведен пример аннотации, где в качестве значения для элемента `assignedTo` по умолчанию выбирается символьная строка `"[none]"`.

```
@BugReport(severity=10)
```

ВНИМАНИЕ! Устанавливаемые по умолчанию значения не хранятся вместе с аннотацией, а вычисляются динамически. Так, если заменить значение, устанавливаемое по умолчанию в элементе **assignedTo**, строковым значением **"[]"** и скомпилировать интерфейс **BugReport** заново, в аннотации **@BugReport(severity=10)** будет использоваться новое устанавливаемое по умолчанию значение — даже в тех файлах классов, которые были скомпилированы до изменения этого значения.

Аннотации можно упростить с помощью двух специальных сокращений. Так, если элементы не указаны, потому что они просто отсутствуют в аннотации или все они принимают значения, устанавливаемые по умолчанию, то круглые скобки не требуются. Например, аннотация

```
@BugReport
```

равнозначна такой аннотации:

```
@BugReport(assignedTo="[none]", severity=0)
```

Подобная аннотация называется *маркерной*.

Другим специальным сокращением является *однозначная аннотация*. Если элемент аннотации имеет специальное имя `value` и больше никаких элементов не указано, то имя элемента и знак = могут быть опущены. Так, если определить интерфейс аннотаций `ActionListenerFor`, упоминавшийся в предыдущем разделе, следующим образом:

```
public @interface ActionListenerFor
{
  String value();
}
```

то сами аннотации можно записать так:

```
@ActionListenerFor("yellowButton")
```

а не так, как показано ниже.

```
@ActionListenerFor(value="yellowButton")
```

У каждого элемента может быть несколько аннотаций:

```
@Test
@BugReport(showStopper=true, reportedBy="Joe")
public void checkRandomInsertions()
```

Если автор аннотации объявил ее повторяющейся, такую аннотацию можно повторять неоднократно, как показано в следующем примере кода:

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
public void checkRandomInsertions()
```

НА ЗАМЕТКУ! Аннотации вычисляются компилятором, и поэтому значения всех элементов аннотаций должны быть представлены константами, обрабатываемыми во время компиляции, как в следующем примере кода:

```
@BugReport(showStopper=true, assignedTo="Harry",
           testCase=MyTestCase.class,
           status=BugReport.Status.CONFIRMED, . . .)
```

ВНИМАНИЕ! Элемент аннотации вообще не может принимать пустое значение **`null`**. Даже по умолчанию не допускается устанавливать в нем пустое значение **`null`**. Это не очень удобно, поскольку для установки по умолчанию придется выбирать другие значения вроде `""` или **`Void.class`**.

Если в качестве значения элемента аннотации указывается массив, значения элементов такого массива заключаются в фигурные скобки:

```
@BugReport(. . ., reportedBy={"Harry", "Carl"})
```

Если же элемент аннотации принимает единственное значение, фигурные скобки можно опустить:

```
// допускается и подобно {"Joe"}:
@BugReport(. . ., reportedBy="Joe")
```

В качестве элемента аннотации может служить какая-нибудь другая аннотация, что дает возможность создавать довольно сложные аннотации, как показано в приведенном ниже примере.

```
@BugReport(ref=@Reference(id="3352627"), . . .)
```

НА ЗАМЕТКУ! Внедрение циклических зависимостей в аннотациях считается ошибкой. Например, вследствие того, что аннотация **BugReport** содержит элемент типа **Reference**, аннотация **Reference** не может содержать элемент типа **BugReport**.

8.4.3. Аннотирование объявлений

Аннотации могут встречаться и во многих других местах прикладного кода. Эти места можно разделить на две категории: *объявления* и *места употребления типов данных*. Аннотации могут появляться в объявлениях следующих элементов кода.

- Пакеты.
- Классы (включая и перечисления).
- Методы.
- Конструкторы.
- Переменные экземпляра (включая и константы перечислимого типа).
- Локальные переменные.
- Переменные параметров.
- Параметры типа.

В объявлениях классов и интерфейсов аннотации указываются перед ключевым словом `class` или `interface` следующим образом:

```
@Entity public class User { ... }
```

В объявлениях переменных аннотации указываются перед типом переменной таким образом:

```
@SuppressWarnings("unchecked") List<User> users = ...;
public User getUser(@Param("id") String userId)
```

Параметр типа в обобщенном классе или методе может быть аннотирован так:

```
public class Cache<@Immutable V> { ... }
```

Пакет аннотируется в отдельном файле `package-info.java`, который содержит только операторы объявления и импорта пакета с предшествующими аннотациями, как показано ниже. Обратите внимание на то, что оператор `import` следует *после* оператора `package`, в котором объявляется пакет.

```
/**
  Документирующий комментарий на уровне пакета
*/
@GPL(version="3")
package com.horstmann.corejava;
import org.gnu.GPL;
```

НА ЗАМЕТКУ! Аннотации всех локальных переменных отбрасываются при компилировании класса. Следовательно, они могут быть обработаны только на уровне исходного кода. Аналогично, аннотации пакетов не сохраняются вне уровня исходного кода.

8.4.4. Аннотирование в местах употребления типов данных

Аннотация в объявлении предоставляет некоторые сведения об объявляемом элементе кода. Так, в следующем примере кода аннотацией утверждается, что параметр `userId` объявляемого метода не является пустым:

```
public User getUser(@NonNull String userId)
```

НА ЗАМЕТКУ! Аннотация **`@NonNull`** является частью каркаса Checker Framework (**`https://checkerframework.org/`**). С помощью этого каркаса можно включать утверждения в прикладную программу, например, утверждение, что параметр не является пустым или относится к типу **`String`** и содержит регулярное выражение. В таком случае инструментальное средство статистического анализа проверит достоверность утверждений в данном теле исходного кода.

Теперь допустим, что имеется параметр типа `List<String>` и требуется каким-то образом указать, что все символьные строки не являются пустыми. Именно здесь и пригодятся аннотации в местах употребления типов данных. Такую аннотацию достаточно указать перед аргументом типа следующим образом:

```
List<@NonNull String>
```

Подобные аннотации можно указывать в следующих местах употребления типов данных.

- Вместе с аргументами обобщенного типа: `List<@NonNull String>`, `Comparator.<@NonNull String> reverseOrder()`.
- В любом месте массива: `@NonNull String[][] words` (элемент массива `words[i][j]` не является пустым), `String @NonNull [][] words` (массив `words` не является пустым), `String[] @NonNull [] words` (элемент массива `words[i]` не является пустым).
- В суперклассах и реализуемых интерфейсах: `class Warning extends @Localized Message`.
- В вызовах конструкторов: `new @Localized String(...)`.
- Во вложенных типах: `Map.@Localized Entry`.
- В операции приведения и проверки типов `instanceof`: `(@Localized String) text`, `if (text instanceof @Localized String)`. (Аннотации служат для употребления только внешними инструментальными средствами и не оказывают никакого влияния на поведение операции приведения и проверки типов `instanceof`.)
- В местах указания исключений: `public String read() throws @Localized IOException`.
- Вместе с метасимволами подстановки и ограничениями типов: `List<@Localized ? extends Message>`, `List<? Extends @Localized Message>`.
- В ссылках на методы и конструкторы: `@Localized Message::getText`.

Однако имеются некоторые места употребления типов данных, где аннотации не допускаются. Ниже приведены характерные примеры.

```
@NonNull String.class // ОШИБКА: литерал класса не
                      // подлежит аннотированию!
import java.lang.@NonNull String; // ОШИБКА: импорт не
                                  // подлежит аннотированию!
```

Аннотации можно размещать до или после других модификаторов доступа вроде `private` и `static`. Обычно (хотя и не обязательно) аннотации в местах употребления типов размещаются после других модификаторов доступа, тогда как аннотации в объявлениях — перед другими модификаторами доступа. Соответствующие примеры приведены ниже.

```
// Аннотация в месте употребления типа данных:
private @NonNull String text;
// Аннотация в объявлении переменной:
@Id private String userId;
```

НА ЗАМЕТКУ Автор аннотации должен указать место, в котором может появиться конкретная аннотация. Если аннотация допускается как в объявлении переменной, так и в месте употребления типа данных, а применяется в объявлении переменной, то она указывается и в том и в другом месте. Рассмотрим в качестве примера следующее объявление метода:

```
public User getUser(@NonNull String userId)
```

Если аннотацию **`@NonNull`** можно применять как в параметрах, так и в местах употребления типов данных, то параметр **`userId`** аннотируется, а тип параметра обозначается как **`@NonNull String`**.

8.4.5. Аннотирование по ссылке `this`

Допустим, требуется аннотировать параметры, которые не изменяются методом:

```
public class Point {
  public boolean equals(@ReadOnly Object other) { ... }
}
```

В таком случае инструментальное средство, обрабатывающее данную аннотацию, после анализа вызова

```
p.equals(q)
```

посчитает, что параметр `q` не изменился. А как насчет ссылки `p`? При вызове данного метода переменная `this` привязывается к ссылке `p`. Но ведь переменная `this` вообще не объявляется, а следовательно, не может быть аннотирована.

На самом деле эту переменную можно объявить с помощью редко употребляемой разновидности синтаксиса, чтобы ввести аннотацию следующим образом:

```
public class Point {
  public boolean equals(@ReadOnly Point this,
                        @ReadOnly Object other) { ... }
}
```

Первый параметр в приведенном выше примере кода называется *параметром получателя*. Он должен непременно называться `this`. Его тип относится к тому классу, объект которого создается.

НА ЗАМЕТКУ! Параметром получателя можно снабдить *только* методы, но *не* конструкторы. По существу, ссылка **this** в конструкторе не является объектом данного типа до тех пор, пока конструктор не завершится. Напротив, аннотация, размещаемая в конструкторе, описывает конструируемый объект.

Конструктору внутреннего класса передается другой скрытый параметр, а именно: ссылка на объект объемлющего класса. Этот параметр также можно указать явным образом:

```
static class Sequence {
  private int from;
  private int to;

  class Iterator implements java.util.Iterator<Integer> {
    private int current;

    public Iterator(@ReadOnly Sequence Sequence.this) {
      this.current = Sequence.this.from;
    }
    ...
  }
  ...
}
```

Этот параметр именуется таким же образом, как и при ссылке на него: *ОбъемлющийКласс*`.this`. А его тип относится к объемлющему классу.

8.5. Стандартные аннотации

В пакетах `java.lang`, `java.lang.annotation` и `javax.annotation` определен целый ряд интерфейсов аннотаций. Четыре из них определяют мета-аннотации, описывающие поведение интерфейсов аннотаций, а остальные — обычные аннотации, которые разработчики могут применять для аннотирования элементов в своем исходном коде. Все эти аннотации вкратце перечислены в табл. 8.2 и более подробно будут описаны в двух последующих разделах.

Таблица 8.2. Стандартные аннотации

Интерфейс аннотаций	Применение	Назначение
Deprecated	Все элементы кода	Аннотирует элемент кода как не рекомендуемый для применения
SuppressWarnings	Все элементы кода, кроме пакетов и аннотаций	Подавляет предупреждения указанного типа
SafeVarargs	Методы и конструкторы	Утверждает, что аргументы переменной длины безопасны для употребления

Окончание табл. 8.2

Интерфейс аннотаций	Применение	Назначение
`Override`	Методы	Проверяет, переопределяет ли данный метод соответствующий метод из суперкласса
`FunctionalInterface`	Интерфейсы	Обозначает интерфейс как функциональный с единственным абстрактным методом
`PostConstruct`, `PreDestroy`	Методы	Обозначает, что аннотированный метод должен вызываться сразу же после создания или непосредственно перед удалением
`Resource`	Классы, интерфейсы, методы, поля	Если это класс или интерфейс, то аннотирует его как ресурс для применения в каком-нибудь другом месте. Если это метод или поле, то аннотирует его как ресурс для внедрения
`Resources`	Классы, интерфейсы	Аннотирует массив ресурсов
`Generated`	Все элементы кода	Аннотирует элемент как исходный код, сгенерированный каким-нибудь инструментальным средством
`Target`	Аннотации	Обозначает элементы, к которым может быть применена данная аннотация
`Retention`	Аннотации	Обозначает, как долго должна сохраняться данная аннотация
`Documented`	Аннотации	Обозначает, что данная аннотация должна быть включена в документацию на аннотируемые элементы кода
`Inherited`	Аннотации	Обозначает, что если данная аннотация применяется к классу, то она будет автоматически наследоваться всеми его подклассами
`Repeatable`	Аннотации	Обозначает, что данную аннотацию можно неоднократно применять к одному и тому же элементу

8.5.1. Аннотации для компиляции

Аннотация `@Deprecated` может присоединяться к любым элементам, применение которых впредь не рекомендуется. В таком случае компилятор будет выдавать соответствующее предупреждение, если нерекомендуемый элемент все-таки применяется в исходном коде. Эта аннотация имеет такое же назначение, как и дескриптор `@deprecated` в документирующих комментариях формата Javadoc.

НА ЗАМЕТКУ! Утилита **`jdeprscan`**, входящая в состав комплекта JDK, позволяет просматривать архивные JAR-файлы на наличие не рекомендованных к употреблению элементов кода.

Аннотация `@SuppressWarnings` указывает компилятору подавлять предупреждения определенного типа, как показано в приведенной ниже строке кода.

```
@SuppressWarnings("unchecked")
```

Аннотация `@Override` применяется только к методам. Компилятор проверяет, чтобы метод с такой аннотацией действительно переопределял соответствующий метод из суперкласса. Так, если сделать приведенное ниже объявление, компилятор выдаст ошибку, поскольку метод `equals()` не переопределяет аналогичный метод `equals()` из класса `Object`. Этот метод имеет параметр типа `Object`, а не `MyClass`.

```
public MyClass
{
  @Override public boolean equals(MyClass other);
  . . .
}
```

Аннотация `@Generated` предназначена для использования инструментальными средствами генерирования кода. Любой генерируемый исходный код может снабжаться аннотациями, чтобы отличаться от кода, предоставляемого программистом. Например, редактор кода может скрывать сгенерированный код, а генератор кода — удалять более старые версии сгенерированного кода. Каждая такая аннотация должна содержать однозначный идентификатор для генератора кода. Символьные строки с датами (в формате ISO 8601) и комментариями указывать необязательно. Ниже приведен характерный пример аннотации для генерирования кода.

```
@Generated("com.horstmann.beanproperty",
           "2008-01-04T12:08:56.235-0700");
```

8.5.2. Аннотации для управления ресурсами

Аннотации `@PostConstruct` и `@PreDestroy` применяются в таких средах, которые управляют жизненным циклом объектов, например, в веб-контейнерах и серверах приложений. Методы, снабжаемые такими аннотациями, должны вызываться сразу же после создания объекта или непосредственно перед его удалением.

Аннотация `@Resource` предназначена для внедрения ресурсов. Рассмотрим в качестве примера веб-приложение, получающее доступ к базе данных. Разумеется, сведения о получении доступа к базе данных не должны жестко кодироваться в таком приложении. Вместо этого у веб-контейнера должен быть какой-то пользовательский интерфейс для установки параметров подключения, а также имя JNDI для источника данных. Сослаться на источник данных в веб-приложении можно следующим образом:

```
@Resource(name="jdbc/mydb")
private DataSource source;
```

При построении объекта, содержащего такое поле, веб-контейнер внедрит ссылку на указанный источник данных.

8.5.3. Мета-аннотации

Мета-аннотация `@Target` применяется к аннотации, ограничивая те элементы, которые должны быть снабжены данной аннотацией, как показано в приведенном ниже примере.

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface BugReport
```

В табл. 8.3 перечислены все возможные значения элементов данной мета-аннотации. Они относятся к перечислимому типу `ElementType`. Указывать можно любое количество типов элементов, заключая их в фигурные скобки.

Таблица 8.3. Типы элементов для мета-аннотации `@Target`

Тип элемента	Применение аннотации
ANNOTATION_TYPE	Объявления типов аннотаций
PACKAGE	Пакеты
TYPE	Классы (включая перечисления) и интерфейсы (включая типы аннотаций)
METHOD	Методы
CONSTRUCTOR	Конструкторы
FIELD	Поля (включая константы перечислимого типа)
PARAMETER	Параметры методов или конструкторов
LOCAL_VARIABLE	Локальные переменные
TYPE_PARAMETER	Параметры типа
TYPE_USE	Места употребления типов данных

Аннотацией без ограничений, накладываемых мета-аннотацией `@Target`, можно снабдить любой элемент кода. Компилятор проверяет, чтобы аннотацией снабжался только разрешенный элемент кода. Так, если снабдить аннотацией `@BugReport` поле, компилятор выдаст во время компиляции соответствующую ошибку.

Мета-аннотация `@Retention` обозначает, насколько долго должна сохраняться аннотация. Указать можно не больше одного значения из перечисленных в табл. 8.4. По умолчанию устанавливается значение константы `RetentionPolicy.CLASS`.

Таблица 8.4. Правила сохраняемости для мета-аннотации `@Retention`

Правило сохраняемости	Описание
SOURCE	Аннотации не включаются в файлы классов
CLASS	Аннотации включаются в файлы классов, но виртуальной машине не нужно их загружать
RUNTIME	Аннотации включаются в файлы классов и загружаются виртуальной машиной. Они доступны через прикладной интерфейс API для рефлексии

В упоминавшемся ранее примере реализации интерфейса аннотаций из листинга 8.8 аннотация `@ActionListenerFor` была объявлена со значением константы `RetentionPolicy.RUNTIME`, поскольку для обработки аннотаций в данном примере применялась рефлексия. В двух последующих разделах приведены примеры обработки аннотаций как на уровне исходного кода, так и на уровне файлов классов.

Мета-аннотация `@Documented` выдает подсказку о средствах документирования в формате Javadoc. Документируемые аннотации следует рассматривать в целях документирования таким же образом, как и другие модификаторы, подобные `protected` или `static`. О применении остальных аннотаций в документации не упоминается. Допустим, аннотация `@ActionListenerFor` объявляется как документируемая следующим образом:

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
```

В таком случае документация на каждый аннотированный метод будет содержать эту аннотацию, как показано на рис. 8.2. Если же аннотация оказывается временной (как, например, `@BugReport`), то документировать ее применение вряд ли стоит.

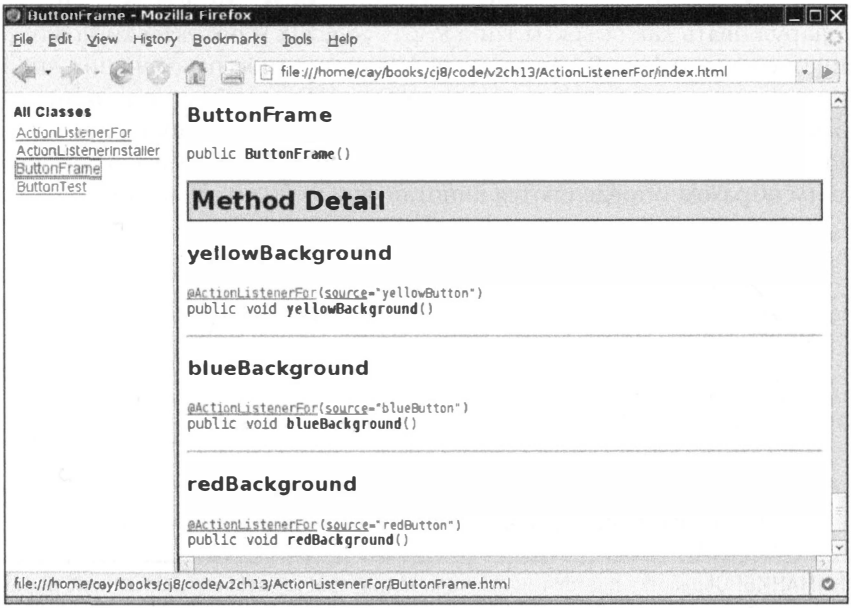


Рис. 8.2. Документируемые аннотации

НА ЗАМЕТКУ! Аннотацию вполне допустимо применять к самой себе. Например, аннотация **`@Documented`** аннотируется как **`@Documented`**. Поэтому в документации формата Javadoc на такие аннотации указано, являются ли они документируемыми.

Мета-аннотация `@Inherited` применяется только к аннотациям классов. Когда в класс вводится наследуемая аннотация, все его подклассы автоматически снабжаются точно такой же аннотацией. Это позволяет легко создавать аннотации, действующие таким же образом, как и маркерные интерфейсы вроде `Serializable`.

На самом деле аннотация `@Serializable` будет более уместной, чем маркерный интерфейс `Serializable` без единого метода. Класс является сериализируемым из-за наличия во время выполнения поддержки для чтения и записи его полей, а не из-за каких-то принципов объектно-ориентированного проектирования. Аннотация описывает данное обстоятельство намного лучше, чем наследование интерфейсов, ведь интерфейс `Serializable` появился еще в версии JDK 1.1, т.е. намного раньше, чем аннотации.

Допустим, для указания на то, что объекты класса могут сохраняться в базе данных, определяется наследуемая аннотация `@Persistent`. В таком случае все подклассы этого класса будут автоматически аннотироваться как сохраняемые:

```
@Inherited @interface Persistent { }
@Persistent class Employee { . . . }
class Manager extends Employee { . . . }
    // также @Persistent
```

При поиске объектов, сохраняемых в базе данных, механизм сохраняемости будет обнаруживать как объекты типа `Employee`, так и объекты типа `Manager`.

Начиная с версии Java 8 допускается неоднократное применение аннотации одного и того же типа к отдельному элементу. Ради обратной совместимости разработчикам повторяющейся аннотации пришлось предоставить *контейнерную аннотацию*, содержащую повторяющиеся аннотации в массиве. Ниже показано, каким образом определяются аннотация `@TestCase` и ее контейнер.

```
@Repeatable(TestCases.class)
@interface TestCase {
  String params();
  String expected();
}

@interface TestCases {
  TestCase[] value();
}
```

Всякий раз, когда пользователь предоставляет две или более аннотации `@TestCase`, они автоматически заключаются в оболочку аннотации `@TestCases`.

ВНИМАНИЕ! Обработка повторяющихся аннотаций требует особого внимания. Если вызвать метод **`getAnnotation()`** для поиска повторяющейся аннотации, которая на самом деле не повторялась, то и в этом случае может быть получено пустое значение **`null`**. Объясняется это тем, что повторяющиеся аннотации были заключены в оболочку контейнерной аннотации.

В таком случае следует вызвать метод **`getAnnotationsByType()`**, где просматривается контейнер и предоставляется массив повторяющихся аннотаций. Если бы имелась только одна аннотация, она была бы получена в массиве единичной длины. Имея в своем распоряжении данный метод, можно вообще не беспокоиться о контейнерной аннотации.

8.6. Обработка аннотаций на уровне исходного кода

В предыдущем разделе было показано, каким образом аннотации анализируются в выполняющейся программе. Еще одним примером применения аннотаций служит автоматическая обработка исходных файлов для получения

дополнительного исходного кода, файлов конфигурации, сценариев и вообще всего, что можно сгенерировать.

8.6.1. Процессоры аннотаций

Обработка аннотаций встроена в компилятор Java. Во время компиляции *процессоры аннотаций* можно вызывать по следующей команде:

```
javac -processor ИмяКлассаПроцессора1, ИмяКлассаПроцессора2,...
                 Исходные_файлы
```

Компилятор обнаруживает аннотации в исходных файлах. Каждый процессор аннотаций выполняется по очереди с учетом тех аннотаций, к которым он проявил интерес. Если процессор аннотаций создает новый исходный файл, то данный процесс повторяется. Как только все исходные файлы будут обработаны, они компилируются.

НА ЗАМЕТКУ Процессор аннотаций может только формировать новые исходные файлы, но не может изменять уже имеющиеся.

Процессор аннотаций реализует интерфейс `Processor`, как правило, расширяя класс `AbstractProcessor`. При этом нужно указать, какие именно аннотации поддерживаются процессором. В данном случае это следующие аннотации:

```
@SupportedAnnotationTypes("com.horstmann.annotations.ToString")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class ToStringAnnotationProcessor
       extends AbstractProcessor
{
  @Override
  public boolean process(
         Set<? extends TypeElement> annotations,
         RoundEnvironment currentRound)
  {
    ...
  }
}
```

Процессору могут потребоваться конкретные типы аннотаций, метасимволы подстановки вроде `"com.horstmann.*"` (т.е. все аннотации из пакета `com.horstmann` и любых его подпакетов) или даже `"*"` (т.е. все аннотации вообще). Метод `process()` вызывается один раз в каждом цикле обработки со всеми аннотациями, обнаруженными в любых файлах при выполнении данного цикла, а также со ссылкой на интерфейс `RoundEnvironment`, содержащей сведения о текущем цикле обработки.

8.6.2. Прикладной интерфейс API модели языка

Для анализа аннотаций на уровне исходного кода служит прикладной интерфейс API *модели языка*. В отличие от прикладного интерфейса API для рефлексии, представляющего классы и методы на уровне виртуальной машины, прикладной интерфейс API модели языка позволяет анализировать программу на Java по правилам языка Java.

Компилятор получает дерево, узлами которого являются экземпляры классов, реализующих интерфейс `javax.lang.model.element.Element`, и производные от него интерфейсы `TypeElement`, `VariableElement`, `ExecutableElement` и т.д. Они служат статическими (на стадии компиляции) аналогами классов рефлексии `Class`, `Field`/`Parameter`, `Method`/`Constructor`.

Не вдаваясь в подробности прикладного интерфейса API модели языка, перечислим главные его особенности, о которых следует знать, приступая к обработке аннотаций.

- Интерфейс `RoundEnvironment` предоставляет все элементы кода, помеченные конкретной аннотацией. Для этой цели вызывается следующий метод:

```
Set<? extends Element> getElementsAnnotatedWith(
                       Class<? extends Annotation> a)
```

- Эквивалентом интерфейса `AnnotateElement` для обработки аннотаций на уровне исходного кода является интерфейс `AnnotatedConstruct`. Для получения обычных или повторяющихся аннотаций из отдельного аннотированного класса служат следующие методы:

```
A getAnnotation(Class<A> annotationType)
A[] getAnnotationsByType(Class<A> annotationType)
```

- Интерфейс `TypeElement` представляет класс или интерфейс, а метод `getEnclosedElements()` получает список его полей и методов.
- В результате вызова метода `getSimpleName()` по ссылке типа `Element` или метода `getQualifiedName()` по ссылке типа `TypeElement` получается объект типа `Name`, который может быть преобразован в символьную строку методом `toString()`.

8.6.3. Генерирование исходного кода с помощью аннотаций

В качестве примера рассмотрим применение аннотаций с целью упростить реализацию методов типа `toString`. Такие методы нельзя ввести в исходные классы. Ведь процессоры аннотаций способны производить только новые классы, а не изменять уже имеющиеся. Следовательно, все подобные методы должны быть введены в служебный класс `ToStrings` следующим образом:

```
public class ToStrings {
  public static String toString(Point obj) {
    Сгенерированный код
  }
  public static String toString(Rectangle obj) {
    Сгенерированный код
  }
  ...
  public static String toString(Object obj) {
    return Objects.toString(obj);
  }
}
```

В данном случае применять рефлексию не требуется, поэтому аннотируются методы доступа, но не поля:

```
@ToString
public class Rectangle {
  ...
  @ToString(includeName=false) public Point getTopLeft()
  { return topLeft; }
  @ToString public int getWidth() { return width; }
  @ToString public int getHeight() { return height; }
}
```

И тогда процессор аннотаций должен сгенерировать следующий исходный код:

```
public static String toString(Rectangle obj) {
  StringBuilder result = new StringBuilder();
  result.append("Rectangle");
  result.append("[");
  result.append(toString(obj.getTopLeft()));
  result.append(",");
  result.append("width=");
  result.append(toString(obj.getWidth()));
  result.append(",");
  result.append("height=");
  result.append(toString(obj.getHeight()));
  result.append("]");
  return result.toString();
}
```

Шаблонный код выделен выше обычным шрифтом. Ниже приведен набросок метода, получающего метод `toString()` для класса с заданным параметром типа `TypeElement`.

```
private void writeToStringMethod(
             PrintWriter out, TypeElement te) {
  String className = te.getQualifiedName().toString();
  Вывести заголовок метода и объявление построителя
  символьных строк
  ToString ann = te.getAnnotation(ToString.class);
  if (ann.includeName())
    Вывести код для ввода имени класса
  for (Element c : te.getEnclosedElements()) {
    ann = c.getAnnotation(ToString.class);
    if (ann != null) {
      if (ann.includeName())
        Вывести код для ввода имени поля
      Вывести код для присоединения
      метода toString(obj.ИмяМетода())
    }
  }
  Вывести код для возврата символьной строки
}
```

Ниже приведен набросок метода `process()` из процессора аннотаций. В этом методе создается исходный файл для вспомогательного класса, а также выводится заголовок класса и по одному методу для каждого аннотируемого класса.

```
public boolean process(
               Set<? extends TypeElement> annotations,
               RoundEnvironment currentRound) {
  if (annotations.size() == 0) return true;
```

```
  try
  {
    JavaFileObject sourceFile = processingEnv.getFiler()
            .createSourceFile(
                    "com.horstmann.annotations.ToStrings");
    try (PrintWriter out =
         new PrintWriter(sourceFile.openWriter()))
    {
      Вывести код для пакета и класса
      for (Element e : currentRound
              .getElementsAnnotatedWith(ToString.class))
      {
        if (e instanceof TypeElement)
        {
          TypeElement te = (TypeElement) e;
          writeToStringMethod(out, te);
        }
      }
      Вывести код для метода toString(Object)
    } catch (IOException ex)
    {
      processingEnv.getMessager().printMessage(
                    Kind.ERROR, ex.getMessage());
    }
  }
  return true;
}
```

За более подробными сведениями обращайтесь к примерам кода, сопровождающего данную книгу. Следует, однако, иметь в виду, что метод `process()` вызывается в последующих циклах обработки аннотаций с пустым списком аннотаций. И тогда происходит немедленный возврат из данного метода, чтобы не создавать исходный файл дважды.

Сначала скомпилируйте процессор аннотаций, а затем скомпилируйте и выполните тестовую программу, введя следующие команды:

```
javac sourceAnnotations/ToStringAnnotationProcessor.java
javac -processor sourceAnnotations \
      .ToStringAnnotationProcessor rect/*.java
java rect.SourceLevelAnnotationDemo
```

СОВЕТ. Чтобы просмотреть циклы обработки аннотаций, выполните команду **javac** с параметром **-XprintRounds**. В итоге на экран будет выведен результат, аналогичный следующему:

```
Round 1:
input files: {ch11.sec05.Point, ch11.sec05.Rectangle,
  ch11.sec05.SourceLevelAnnotationDemo}
  annotations: [com.horstmann.annotations.ToString]
  last round: false
Round 2:
  input files: {com.horstmann.annotations.ToStrings}
  annotations: []
  last round: false
Round 3:
  input files: {}
  annotations: []
  last round: true
```

В данном примере было продемонстрировано, каким образом инструментальные средства могут собирать аннотации из исходных файлов для получения других файлов. Формируемые в итоге файлы совсем не обязательно должны быть исходными. Процессоры аннотаций могут сформировать дескрипторы XML-разметки, файлы свойств, сценарии командного процессора, документацию в формате HTML и пр.

НА ЗАМЕТКУ! Были предложения использовать аннотации, чтобы еще больше сократить объем рутинной работы. В самом деле, было бы замечательно, если бы тривиальные методы получения и установки генерировались автоматически. Например, аннотация

```
@Property private String title;
```

могла бы автоматически генерировать приведенные ниже методы.

```
public String getTitle() { return title; }
public void setTitle(String title) { this.title = title; }
```

Но эти методы должны быть введены *в один и тот же* класс. Для этого потребуется редактирование исходного файла, а не только генерирование еще одного файла, что выходит за пределы возможностей обработчиков аннотаций. С этой целью можно было бы создать другое инструментальное средство, но оно уже не вписывалось бы в рамки основного назначения аннотаций. Ведь аннотация предназначена для описания элемента кода, а *не* в качестве директивы для добавления или изменения кода.

8.7. Конструирование байт-кодов

Как пояснялось ранее, аннотации могут обрабатываться как во время выполнения, так и на уровне исходного кода. Но существует еще и третий способ обработки аннотаций на уровне байт-кода. Если аннотации не удаляются на уровне исходного кода, они присутствуют в файлах классов. Формат этих файлов документирован (см. по адресу `https://docs.oracle.com/javase/specs/jvms/se10/html`). Это довольно сложный формат, и поэтому обрабатывать файлы классов без специальных библиотек было бы совсем не просто. К их числу относится библиотека ASM, доступная по адресу `http://asm.ow2.org`.

8.7.1. Модификация файлов классов

В этом разделе будет показано, как пользоваться библиотекой ASM для добавления в аннотированные методы протокольных сообщений. Так, если метод снабжен следующей аннотацией:

```
@LogEntry(logger=ИмяРегистратора)
```

то в начале этого метода вводятся байт-коды для оператора

```
Logger.getLogger(ИмяРегистратора)
      .entering(ИмяКласса, ИмяМетода);
```

Если, например, снабдить такой аннотацией метод `hashCode()` из класса `Item` следующим образом:

```
@LogEntry(logger="global") public int hashCode()
```

то при каждом вызове этого метода будет выводиться приблизительно такое сообщение:

```
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
```

Чтобы добиться такого результата, необходимо выполнить следующие действия.

1. Загрузить байт-коды в файл класса.
2. Определить местонахождение всех методов.
3. Выполнить для каждого метода проверку на наличие в нем аннотации `LogEntry`.
4. Если такая аннотация присутствует, ввести байт-коды для следующих инструкций в начале метода:

```
ldc ИмяРегистратора
invokestatic java/util/logging/Logger.getLogger:
    (Ljava/lang/String;)Ljava/util/logging/Logger;
ldc ИмяКласса
ldc ИмяМетода
invokevirtual java/util/logging/Logger.entering:
    (Ljava/lang/String;Ljava/lang/String;)V
```

Вставка этих байт-кодов может показаться на первый взгляд сложной задачей, но библиотека ASM существенно упрощает ее. Не вдаваясь в подробности анализа и вставки байт-кодов, обратимся к конкретному примеру программы из листинга 8.9. В этой программе редактируется файл классов и вставляется вызов регистратора в начале всех методов, снабженных аннотацией `LogEntry`.

Ниже показано, каким образом инструкции для протоколирования вводятся в исходный файл `Item.java`, представленный в листинге 8.10, где ***asm*** — каталог, в котором установлена библиотека ASM.

```
javac set/Item.java
javac -classpath .:asm/lib/\*
      bytecodeAnnotations/EntryLogger.java
java -classpath .:asm/lib/\*
     bytecodeAnnotations.EntryLogger set.Item
```

Попробуйте выполнить следующую команду до и после изменения файла класса `Item`:

```
javap -c set.Item
```

В итоге инструкции для протоколирования должны быть вставлены в начале методов `hashCode()`, `equals()` и `compareTo()`, как показано ниже.

```
public int hashCode();
  Code:
    0: ldc    #85; // String global
    2: invokestatic #80;
      // Method java/util/logging/Logger.getLogger:
      //    (Ljava/lang/String;)Ljava/util/logging/Logger;
    5: ldc    #86; // String Item
    7: ldc    #88; // String hashCode
```

```
    9: invokevirtual    #84;
      // Method java/util/logging/Logger.entering:
      //    (Ljava/lang/String;Ljava/lang/String;)V
   12: bipush 13
   14: aload_0
   15: getfield        #2; // Field description:
                           // Ljava/lang/String;
   18: invokevirtual   #15; // Method
                       // java/lang/String.hashCode:()I
   21: imul
   22: bipush 17
   24: aload_0
   25: getfield        #3; // Field partNumber:I
   28: imul
   29: iadd
   30: ireturn
```

Программа `SetTest` из листинга 8.11 вставляет объекты типа `Item` в хеш-множество. Если запустить ее с измененным файлом класса, то появятся протокольные сообщения, аналогичные приведенным ниже.

```
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
May 17, 2016 10:57:59 AM Item hashCode
FINER: ENTRY
May 17, 2016 10:57:59 AM Item equals
FINER: ENTRY
[[description=Toaster, partNumber=1729],
 [description=Microwave, partNumber=4104]]
```

Обратите внимание на вызов метода `equals()` при вставке одного и того же элемента дважды. Данный пример демонстрирует эффективность конструирования байт-кодов. Аннотации используются для ввода в прикладную программу директив, а инструментальное средство редактирования байт-кодов собирает эти директивы и соответственно видоизменяет инструкции виртуальной машины.

Листинг 8.9. Исходный код из файла **bytecodeAnnotations/EntryLogger.java**

```
1   package bytecodeAnnotations;
2
3   import java.io.*;
4   import java.nio.file.*;
5
6   import org.objectweb.asm.*;
7   import org.objectweb.asm.commons.*;
8
9   /**
10   * Вводит инструкции для протоколирования записей
11   * вначале всех методов из класса, снабженного
12   * аннотацией LogEntry
13   * @version 1.21 2018-05-01
14   * @author Cay Horstmann
15   */
```

```
16 public class EntryLogger extends ClassVisitor
17 {
18   private String className;
19
20   /**
21    * Конструирует объект типа EntryLogger,
22    * вставляющий инструкции протоколирования в
23    * аннотированные методы данного класса
24    * @param cg the class
25    */
26   public EntryLogger(ClassWriter writer,
27                      String className)
28   {
29     super(Opcodes.ASM5, writer);
30     this.className = className;
31   }
32
33   public MethodVisitor visitMethod(int access,
34                String methodName, String desc,
35                String signature, String[] exceptions)
36   {
37     MethodVisitor mv = cv.visitMethod(access,
38           methodName, desc, signature, exceptions);
39     return new AdviceAdapter(Opcodes.ASM5, mv, access,
40                              methodName, desc)
41     {
42       private String loggerName;
43
44       public AnnotationVisitor visitAnnotation(
45                        String desc, boolean visible)
46       {
48         return new AnnotationVisitor(Opcodes.ASM5)
49           {
50             public void visit(String name, Object value)
51             {
52               if (desc.equals(
53                       "LbytecodeAnnotations/LogEntry;")
54                    && name.equals("logger"))
55                  loggerName = value.toString();
56             }
57           };
58       }
59
60       public void onMethodEnter()
61         {
62           if (loggerName != null)
63           {
64             visitLdcInsn(loggerName);
65             visitMethodInsn(INVOKESTATIC,
66                "java/util/logging/Logger", "getLogger",
67                "(Ljava/lang/String;)"
68              + "Ljava/util/logging/Logger;", false);
69             visitLdcInsn(className);
70             visitLdcInsn(methodName);
71             visitMethodInsn(INVOKEVIRTUAL,
72                "java/util/logging/Logger", "entering",
```

```
73                "(Ljava/lang/String;Ljava/lang"
74              + "/String;)V", false);
75            loggerName = null;
76          }
77        }
78      };
79    }
80
81    /**
82     * Вводит код регистрации записей в указанный класс
83     * @param args Имя файла класса для вставки кода
84     */
85    public static void main(String[] args)
86          throws IOException
87    {
88      if (args.length == 0)
89      {
90        System.out.println("USAGE: java "
91          + "bytecodeAnnotations.EntryLogger classfile");
92        System.exit(1);
93      }
94      Path path = Paths.get(args[0]);
95      var reader =
96          new ClassReader(Files.newInputStream(path));
97      var writer = new ClassWriter(
98                      ClassWriter.COMPUTE_MAXS
99                      | ClassWriter.COMPUTE_FRAMES);
100     var entryLogger = new EntryLogger(writer,
101     path.toString().replace(".class", "")
102                     .replaceAll("[/\\\\]", "."));
103     reader.accept(entryLogger,
104                   ClassReader.EXPAND_FRAMES);
105     Files.write(Paths.get(args[0]),
106                 writer.toByteArray());
107   }
108 }
```

Листинг 8.10. Исходный код из файла **set/Item.java**

```
1  package set;
2
3  import java.util.*;
4  import bytecodeAnnotations.*;
5
6  /**
7   * Товар с описанием и номенклатурным номером
8   * @version 1.01 2012-01-26
9   * @author Cay Horstmann
10 */
11 public class Item
12 {
13   private String description;
14   private int partNumber;
15
16   /**
```

```
17    * Конструирует объект товара
18    * @param aDescription Описание товара
19    * @param aPartNumber Номенклатурный номер
20    */
21   public Item(String aDescription, int aPartNumber)
22   {
23     description = aDescription;
24     partNumber = aPartNumber;
25   }
26
27   /**
28    * Получить описание данного товара
29    * @return Описание товара
30    */
31   public String getDescription()
32   {
33     return description;
34   }
35
36   public String toString()
37   {
38     return "[description=" + description
39          + ", partNumber=" + partNumber + "]";
40   }
41
42   @LogEntry(logger = "global")
43   public boolean equals(Object otherObject)
44   {
45     if (this == otherObject) return true;
46     if (otherObject == null) return false;
47     if (getClass() != otherObject.getClass())
48         return false;
49     var other = (Item) otherObject;
50     return Objects.equals(description, other.description)
51            && partNumber == other.partNumber;
52   }
53
54   @LogEntry(logger = "global")
55   public int hashCode()
56   {
57     return Objects.hash(description, partNumber);
58   }
59 }
```

Листинг 8.11. Исходный код из файла **set/SetTest.java**

```
1  package set;
2
3  import java.util.*;
4  import java.util.logging.*;
5
6  /**
7   * @version 1.03 2018-05-01
8   * @author Cay Horstmann
9   */
```

```
10 public class SetTest
11 {
12   public static void main(String[] args)
13   {
14     Logger.getLogger("com.horstmann")
15           .setLevel(Level.FINEST);
16     var handler = new ConsoleHandler();
17     handler.setLevel(Level.FINEST);
18     Logger.getLogger("com.horstmann")
19           .addHandler(handler);
20
21     var parts = new HashSet<Item>();
22     parts.add(new Item("Toaster", 1279));
23     parts.add(new Item("Microwave", 4104));
24     parts.add(new Item("Toaster", 1279));
25     System.out.println(parts);
26   }
27 }
```

8.7.2. Модификация байт-кодов во время загрузки

В предыдущем разделе было представлено инструментальное средство, способное редактировать файлы классов. Но внедрение еще одного такого инструментального средства во время компоновки прикладной программы может оказаться непростым делом. Поэтому существует довольно привлекательный альтернативный способ, состоящий в том, чтобы отложить конструирование байт-кодов до стадии *загрузки*, на которой загружаются классы.

В прикладном интерфейсе API для *оснащения инструментальными средствами* имеется перехватчик, позволяющий устанавливать преобразователь байт-кодов. Этот преобразователь должен устанавливаться перед вызовом главного метода программы. Чтобы удовлетворить данному требованию, определяется *агент* (т.е. библиотека, загружаемая для контроля над программой). Код этого агента может выполнять в методе `premain()` операции, требующиеся для инициализации.

Чтобы создать агент, необходимо выполнить следующие действия.

1. Реализовать класс с помощью метода

   ```
   public static void premain(String arg,
                              Instrumentation instr)
   ```

2. Этот метод вызывается при загрузке агента. Агент может получить единственный аргумент командной строки, передаваемый в качестве параметра `arg`. Параметр `instr` можно использовать для установки различных перехватчиков.

3. Создать файл манифеста, в котором устанавливается атрибут `Premain-Class`, например, следующим образом:

   ```
   Premain-Class: bytecodeAnnotations.EntryLoggingAgent
   ```

4. Упаковать код агента и манифест в архивный JAR-файл, например, так, как показано ниже.

```
javac -classpath .:asm/lib/\*
      bytecodeAnnotations/EntryLoggingAgent.java
jar cvfm EntryLoggingAgent.jar \
    bytecodeAnnotations/EntryLoggingAgent.mf \
    bytecodeAnnotations/Entry*.class
```

Чтобы запустить программу на Java вместе с агентом, необходимо указать следующие параметры в командной строке:

```
java -javaagent:JAR-файл_с_агентом=АргументАгента . . .
```

Например, чтобы запустить программу `SetTest` с агентом протоколирования записей, потребуется выполнить приведенные ниже команды, где аргумент **`Item`** обозначает имя класса, который агент должен модифицировать.

```
javac set/SetTest.java
java -javaagent:EntryLoggingAgent.jar=set.Item -classpath \
    .:asm/lib/\* set.SetTest
```

В листинге 8.12 представлен исходный код агента, устанавливающего преобразователь файлов классов. Этот преобразователь сначала проверяет, соответствует ли имя класса аргументу агента. Если оно соответствует, то преобразователь использует класс `EntryLogger`, упоминавшийся в предыдущем разделе, для модификации байт-кодов. Но модифицированные байт-коды не сохраняются в файле. Вместо этого преобразователь возвращает их для загрузки в виртуальную машину (рис. 8.3). Иными словами, данный способ позволяет модифицировать байт-коды в динамическом режиме.

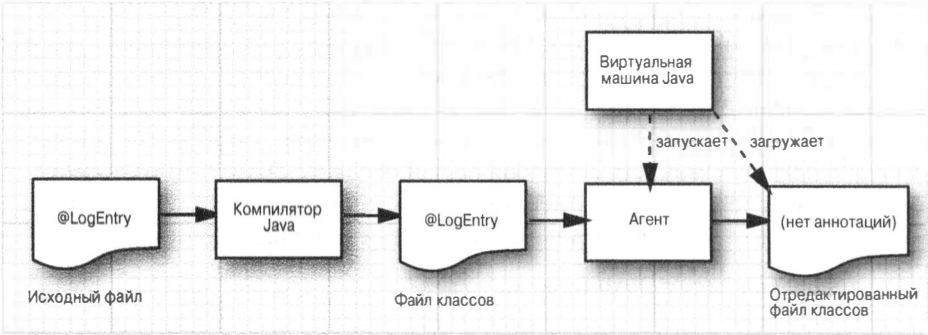


Рис. 8.3. Модификация классов во время загрузки

Листинг 8.12. Исходный код из файла **`bytecodeAnnotations/EntryLoggingAgent.java`**

```
1  package bytecodeAnnotations;
2
3  import java.lang.instrument.*;
4
5  import org.objectweb.asm.*;
6
7  /**
8   * @version 1.11 2018-05-01
9   * @author Cay Horstmann
```

```
 */
public class EntryLoggingAgent
{
  public static void premain(final String arg,
                             Instrumentation instr)
  {
    instr.addTransformer(
          (loader, className, cl, pd, data) ->
      {
        if (!className.replace("/", ".").equals(arg))
          return null;
        var reader = new ClassReader(data);
        var writer = new ClassWriter(
                        ClassWriter.COMPUTE_MAXS
                        | ClassWriter.COMPUTE_FRAMES);
        var el = new EntryLogger(writer, className);
        reader.accept(el, ClassReader.EXPAND_FRAMES);
        return writer.toByteArray();
      });
  }
}
```

Из этой главы вы узнали, как

- вводить аннотации в исходный код программ на Java;
- создавать свои собственные интерфейсы аннотаций;
- реализовывать инструментальные средства, применяющие аннотации.

Здесь продемонстрированы также три методики обработки кода: написание сценариев, компиляция программ на Java и обработка аннотаций. Первые две методики довольно просты. Что же касается создания инструментальных средств обработки аннотаций, то эта методика, безусловно, сложна, и поэтому ею вряд ли рискнут воспользоваться многие разработчики. Однако в этой главе представлены лишь самые основные сведения, необходимые для правильного понимания внутреннего механизма работы типичных инструментальных средств обработки аннотаций. Но, возможно, они вызовут у вас интерес к созданию своих собственных инструментальных средств для этих целей.

В следующей главе речь пойдет уже о совершенно другом предмете: модульной системе на платформе Java, которая стала главным нововведением в версии Java 9, определившим дальнейшее развитие платформы Java.

ГЛАВА 9

Модульная система на платформе Java

В этой главе...

- Понятие модуля
- Именование модулей
- Пример модульной программы "Hello, Modular World!"
- Требования модулей
- Экспорт пакетов
- Модульные архивные JAR-файлы
- Модули и рефлексивный доступ
- Автоматические модули
- Безымянные модули
- Параметры командной строки для переноса прикладного кода
- Переходные и статические требования
- Уточненный экспорт и открытие модулей
- Загрузка служб
- Инструментальные средства для работы с модулями

Важной особенностью ООП является инкапсуляция. Объявление класса состоит из открытого интерфейса и его закрытой реализации. В обязанности класса может входить изменение реализации без воздействия на его пользователей. Модульная система обеспечивает те же преимущества для программирования, но

только в крупном масштабе. Так, в модуле можно создавать выборочно доступные классы и пакеты, чтобы контролировать процесс их развития.

Несколько модульных систем, существующих на платформе Java, опираются на загрузчики классов с целью их изоляции. Тем не менее в версии Java 9 внедрена новая модульная система, которая официально называется Java Platform Module System (Модульная система на платформе Java) и поддерживается компилятором и виртуальной машиной Java. Она предназначена для модуляризации крупной кодовой базы на платформе Java. Но этой системой можно при желании воспользоваться для модуляризации собственных приложений.

Независимо от того, пользуетесь ли вы модулями на платформе Java в своих приложениях, модуляризированная платформа Java все равно будет оказывать на вас влияние. В этой главе показано, как объявляются и применяются модули на платформе Java, как переносить свои приложения для работы как с модуляризированной платформой Java, так и со сторонними модулями.

9.1. Понятие модуля

Основными стандартными блоками в ООП служат классы, обеспечивающие инкапсуляцию. Закрытые функциональные средства класса вроде методов могут быть доступны в коде только по специальному разрешению. В связи с этим возникает вопрос о правах доступа. Так, если закрытая переменная изменилась, то можно выявить всех, кто мог быть тому виной. А если требуется видоизменить закрытое представление, то должно быть заранее известно, на какие методы это окажет воздействие.

Более крупное организационное группирование в Java обеспечивают пакеты, представляющие собой коллекцию классов. Пакеты обеспечивают также соответствующий уровень инкапсуляции. Любое функциональное средство (как открытое, так и закрытое) доступно в пакете только из методов, находящихся в том же самом пакете.

Но в крупных системах такого уровня управления доступом явно недостаточно. Любое функциональное средство, которое является открытым, т.е. доступным за пределами пакета, оказывается доступным повсюду. Допустим, требуется видоизменить или исключить редко используемое функциональное средство. Если оно является открытым, то очень трудно предвидеть последствия такого изменения. На это препятствие натолкнулись в свое время разработчики платформы Java. Комплект JDK стремительно развивался в течение более двадцати лет, но некоторые его функциональные средства явно устарели. Характерным тому примером служит архитектура CORBA. Вряд ли кто-нибудь из разработчиков вспомнит, когда пользовался пакетом `org.omg.corba` в последний раз, хотя он неизменно включался в комплект JDK вплоть до версии Java 10. Начиная с версии Java 11, тем немногим разработчикам, кому этот пакет все еще может понадобиться, придется ввести требующиеся архивные JAR-файлы в свои проекты.

А как насчет пакета `java.awt`? Ведь он вряд ли потребуется в серверном приложении, за исключением класса `java.awt.DataFlavor`, применяемого в реализации SOAP — протокола для веб-служб на основе XML.

Столкнувшись с проблемой растущего как снежный ком объема кода, разработчики платформы Java решили, что им необходим какой-то механизм структурирования, обеспечивающий больший контроль над кодом. С этой целью они проанализировали существующие модульные системы (например, OSGi) и нашли их непригодными для решения стоявшей перед ними задачи. Поэтому они разработали новую систему под названием *Java Platform Module System*, которая теперь входит в состав языка и виртуальной машины Java. Эта модульная система была успешно использована для модуляризации прикладного интерфейса Java API, и вы можете при желании прибегнуть к ней в своих приложениях.

Модуль на платформе Java состоит из следующих частей.

- Коллекция пакетов.
- Дополнительные файлы ресурсов и прочие файлы, включая платформенно-ориентированные библиотеки.
- Список пакетов, доступных в модуле.
- Список всех модулей, от которых зависит данный модуль.

Инкапсуляция и зависимости соблюдаются на платформе Java как во время компиляции, так и при выполнении в виртуальной машине. А зачем вообще рассматривать применение модульной системы на платформе Java в собственных программах, если можно придерживаться традиционного подхода к архивным JAR-файлам, доступным по пути к классам? Применение модульной системы на платформе Java дает следующие преимущества.

1. Строгая инкапсуляция. Доступ к конкретным пакетам можно контролировать, не особенно беспокоясь о поддержании кода, который не предназначается для общего употребления.
2. Надежная конфигурация. Позволяет избежать таких затруднений, связанных с путями к общедоступным классам, как дублирование или отсутствие классов.

Тем не менее имеется ряд вопросов, которые модульная система на платформе Java не в состоянии разрешить. К их числу относится контроль версий модулей, отсутствие возможности указывать конкретную версию требующегося модуля или применять несколько версий модуля в одной и той же прикладной программе. Но для того чтобы воспользоваться этими столь желанными возможностями, придется обратиться к другим механизмам вместо модульной системы на платформе Java.

9.2. Именование модулей

Модуль — это коллекция пакетов. Имена пакетов в модуле не должны быть взаимосвязаны. В состав модуля `java.sql` могут, например, входить пакеты `java.sql`, `javax.sql` и `javax.transaction.xa`. Как следует из этого примера, вполне допустимо, чтобы совпадали имена модулей и входящих в них пакетов.

Аналогично имени пакета, имя модуля состоит из букв, цифр, знаков подчеркивания и точки. Кроме того, между модулями, как и пакетами, не должно быть

иерархической взаимосвязи. Так, если имеется один модуль под именем com.horstmann, а другой — под именем com.horstmann.corejava, то в модульной системе они не должны быть связаны вместе.

При создании модуля, предназначенного для применения в других модулях, очень важно, чтобы его имя было глобально однозначным. При этом предполагается, что имена большинства модулей должны отвечать условным обозначениям в обратном порядке следования имен доменов, как это делается в пакетах.

Модуль проще всего обозначать по имени пакета верхнего уровня, предоставляемого в данном модуле. Например, в фасаде загрузки SLF4J имеется модуль org.slf4j с пакетами org.slf4j, org.slf4j.spi, org.slf4j.event и org.slf4j.helpers.

Такое условное обозначение предотвращает конфликты имен пакетов в модулях. Любой заданный пакет может быть размещен только в одном модуле. Если имена модулей однозначны, а имена пакетов начинаются с имени модуля, то и имена пакетов будут однозначны.

Для обозначения модулей можно употреблять более короткие имена, не предназначенные для применения другими программистами, например, в модуле, содержащем прикладную программу. Именно такой способ именования модулей и демонстрируется в этой главе. Например, модули, которые могут содержать библиотечный код, должны именоваться как com.horstmann.util, а модули, содержащие прикладные программы с классом, в котором имеется метод main(), — более запоминающимися именами вроде v2ch09.hellomod.

НА ЗАМЕТКУ! Имена модулей следует употреблять только в объявлениях самих модулей, но их вообще не стоит употреблять в исходных файлах классов Java. Имена пакетов следует употреблять, как обычно.

9.3. Пример модульной программы "Hello, Modular World!"

Попробуем перенести в модуль традиционную программу, выводящую приветствие "Hello, Modular World!" (Здравствуй, модульный мир!). Для этого нам, прежде всего, понадобится разместить соответствующий класс в пакете, поскольку безымянный пакет не может содержаться в модуле. Ниже показано, как это делается.

```
package com.horstmann.hello;

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

До сих пор никаких особых изменений не произошло. Чтобы создать модуль v2ch09.hellomod, содержащий данный пакет, придется ввести его объявление. Этот модуль размещается в файле module-info.java, расположенном в базовом каталоге, т.е. там же, где и каталог com. Базовый каталог принято именовать таким же образом, как и модуль, как показано ниже.

```
v2ch09.hellomod/
module-info.java
  com/
  horstmann/
    hello/
      HelloWorld.java
```

В файле `module-info.java` содержится приведенное ниже объявление модуля. Объявление данного модуля оказывается пустым потому, что он не только ничего не предоставляет всем остальным модулям, но и ничего не требует от них.

```
module v2ch09.hellomod
{
}
```

Теперь выполним компиляцию, как обычно:

```
javac v2ch09.hellomod/module-info.java \
      v2ch09.hellomod/com/horstmann/hello/HelloWorld.java
```

Файл `module-info.java` не похож на исходный файл Java и, естественно, не может быть класса с именем `module-info`, поскольку имена классов не должны содержать дефиса. Ключевое слово `module`, а также ключевые слова `requires`, `exports` и т.д., употребляемые в приведенных далее примерах кода, относятся к так называемым "ограниченным" ключевым словам, имеющим специальное назначение только в объявлениях модулей. Данный файл компилируется в файл класса `module-info.class`, содержащий определение модуля в двоичной форме.

Чтобы выполнить рассматриваемую здесь программу в виде модульного приложения, следует указать *путь к модулю* аналогично пути к классу, только он должен содержать модули. Кроме того, главный класс должен быть указан в форме *имя_модуля/имя_класса*, как показано ниже.

```
java --module-path v2ch09.hellomod -module \
      v2ch09.hellomod/com.horstmann.hello.HelloWorld
```

Вместо параметров `--module-path` и `-module` в данной команде можно указать их однобуквенные аналоги `-p` и `-m`:

```
java -p v2ch09.hellomod \
     -m v2ch09.hellomod/com.horstmann.hello.HelloWorld
```

Но в любом случае приветствие `"Hello, Modular World!"` появится на экране. Тем самым демонстрируется, что первое наше приложение успешно модуляризировано.

НА ЗАМЕТКУ! В результате компиляции данного модуля будет выдано следующее предупреждение:

```
warning: [module] module name component v2ch09 should avoid terminal
digits[1]
```

Оно предназначено для того, чтобы отвадить разработчиков от желания вводить номера версий в имена модулей. Его можно, конечно, проигнорировать или подавить с помощью следующей аннотации:

[1] `предупреждение: [модуль] в составляющей v2ch09 имени модуля следует избегать употребления оконечных цифр`

```
@SuppressWarnings("module")
module v2ch09.hellomod {
}
```

В данном конкретном отношении объявление **module** подобно объявлению класса. В частности, его можно аннотировать. (Тип аннотации должен иметь ее цель, в данном случае — **ElementType.MODULE**.)

9.4. Требования модулей

Теперь создадим новый модуль `v2ch09.requiremod`, в котором находится класс `JOptionPane`, применяемый в рассматриваемой здесь программе для отображения приветствия `"Hello, Modular World!"`:

```
package com.horstmann.hello;

import javax.swing.JOptionPane;

public class HelloWorld
{
  public static void main(String[] args) {
    JOptionPane.showMessageDialog(null,
                "Hello, Modular World!");
  }
}
```

Компиляция завершится неудачно со следующим сообщением об ошибке:

```
error: package javax.swing is not visible
  (package javax.swing is declared in module java.desktop,
   but module v2ch09.requiremod does not read it)[2]
```

Комплект JDK модуляризирован, и теперь пакет `javax.swing` содержится в модуле `java.desktop`. Таким образом, мы должны объявить свой модуль, как опирающийся на данный модуль.

```
module v2ch09.requiremod {
    requires java.desktop;
}
```

Цель проектирования модульной системы в том и состоит, чтобы требования модулей были явно заданы, а виртуальная машина смогла удовлетворить все эти требования, прежде чем запустить программу на выполнение. В примерах из предыдущего раздела необходимость в явном задании требований модулей не возникала, поскольку мы пользовались в них только пакетами `java.lang` и `java.io` из состава модуля `java.base`, который требуется по умолчанию.

Обратите внимание на то, что в модуле `v2ch09.requiremod` перечисляются только его собственные требования других модулей. В нем требуется модуль `java.desktop`, чтобы пользоваться пакетом `javax.swing`. В самом модуле `java.`

[2] ошибка: пакет javax.swing недоступен
  (пакет javax.swing определяется в модуле java.desktop,
   но в модуле v2ch09.requiremod он не читается)

desktop объявляется, что ему требуются три других модуля, а именно: java.datatransfer, java.prefs и java.xml.

На рис. 9.1 показан *граф модулей*, узлами которого являются отдельные модули, а ребрами, соединяющими узлы стрелками, — объявленные требования или подразумеваемое требование модуля java.base, если никаких требований вообще не объявлено.

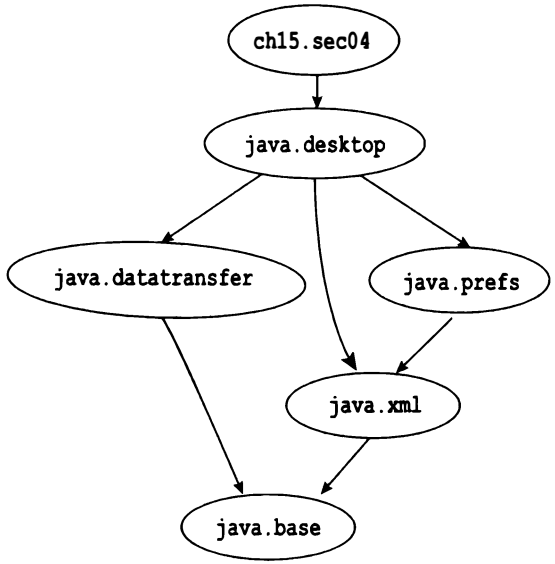


Рис. 9.1. Граф модулей из примера прикладной программы **"Hello, Modular World!"**

Циклы в графе модулей не допускаются. Это означает, что модуль не может прямо или косвенно требовать самого себя.

Права доступа не передаются автоматически от одного модуля к другому. В рассматриваемом здесь примере объявляется, что в модуле java.desktop требуется модуль java.prefs, а в том — модуль java.xml. Но это не дает модулю java.desktop права пользоваться пакетами из модуля java.xml, поскольку он должен быть объявлен явным образом. В математике отношение "требуется" не является транзитивным. В общем, такое поведение желательно, поскольку оно делает требования явными, но, как будет показано в разделе 9.11, его можно иногда смягчить.

НА ЗАМЕТКУ! В сообщении об ошибке, приведенном в начале этого раздела, извещается, что в модуле **v2ch09.requiremod** не прочитан модуль **java.desktop**. В терминах модульной системы на платформе Java это означает, что модуль ***M*** *читает* модуль ***N*** в следующих случаях.

1. Модулю ***M*** требуется модуль ***N***.
2. Модулю ***M*** требуется модуль, которому, в свою очередь, транзитивно требуется модуль ***N*** (см. далее раздел 9.11).
3. Модуль ***N*** является модулем ***M*** или модулем **java.base**.

9.5. Экспорт пакетов

Как было показано в предыдущем разделе, одному модулю требуется другой модуль, если необходимо воспользоваться его пакетами. Но это не делает автоматически доступными все пакеты из требующегося модуля. С помощью ключевого слова `exports` в модуле объявляется, какие из его пакетов являются доступными. Так, в приведенном ниже примере кода представлена часть объявления модуля `java.xml`. Таким образом, в данном пакете становятся доступными одни пакеты, но недоступными другие (например, пакет `jdk.xml.internal`), поскольку они не экспортируются.

```
module java.xml {
    exports javax.xml;
    exports javax.xml.catalog;
    exports javax.xml.datatype;
    exports javax.xml.namespace;
    exports javax.xml.parsers;
    ...
}
```

Когда экспортируется пакет, его открытые (`public`) и защищенные (`protected`) классы, интерфейсы и их члены становятся доступными за пределами данного модуля. (Как обычно, защищенные классы и их члены доступны только в их подклассах.)

Тем не менее пакет, который не экспортируется, все равно недоступен за пределами своего модуля. Это существенное отличие от прежних модулей в Java. В прошлом можно было пользоваться открытыми классами из любого пакета, даже если он и не входил в состав открытого прикладного интерфейса API. Обычно, например, рекомендовалось пользоваться такими классами, как `sun.misc.BASE64Encoder` или `com.sun.rowset.CachedRowSetImpl`, если в открытом прикладном интерфейсе API не предоставлялись соответствующие функциональные возможности.

Ныне пакеты, не экспортированные из прикладного интерфейса API на платформе Java, больше недоступны, поскольку все они содержатся в модулях. В итоге некоторые программы больше не будут выполняться в версии Java 9. Безусловно, это не должно стать неприятной неожиданностью для тех, кто и не собирался опираться на неоткрытые прикладные интерфейсы API.

Воспользуемся операциями экспорта в простой ситуации. С этой целью подготовим модуль `com.horstmann.greet`, в котором экспортируется пакет под тем же самым именем `com.horstmann.greet`, при условии, что модуль, предоставляющий свой код для других модулей, должен обозначаться по имени находящегося в нем пакета верхнего уровня. В этом модуле имеется также пакет `com.horstmann.greet.internal`, который не экспортируется.

Приведенный ниже открытый интерфейс `Greeter` находится в первом пакете.

```
package com.horstmann.greet;

public interface Greeter {
  static Greeter newInstance() {
    return new com.horstmann.greet.internal.GreeterImpl();
```

```
  }

  String greet(String subject);
}
```

Во втором пакете находится приведенный ниже класс, в котором реализуется упомянутый выше интерфейс. Этот класс является открытым, поскольку он доступен в первом пакете.

```
package com.horstmann.greet.internal;

import com.horstmann.greet.Greeter;

public class GreeterImpl implements Greeter {
  public String greet(String subject) {
    return "Hello, " + subject + "!";
  }
}
```

Оба упомянутых выше пакета содержатся в модуле `com.horstmann.greet`, но экспортируется только первый из них, как показано ниже. Поэтому второй пакет недоступен за пределами данного модуля.

```
module com.horstmann.greet {
  exports com.horstmann.greet;
}
```

Перенесем рассматриваемую здесь прикладную программу во второй модуль, где требуется первый модуль:

```
module v2ch09.exportedpkg {
  requires com.horstmann.greet;
}
```

НА ЗАМЕТКУ После оператора **`exports`** указывается имя экспортируемого пакета, тогда как после оператора **`requires`** — имя требуемого модуля.

Интерфейс `Greeter` применяется в рассматриваемой здесь прикладной программе для получения приветствия:

```
package com.horstmann.hello;

import com.horstmann.greet.Greeter;

public class HelloWorld
{
  public static void main(String[] args)
  {
    Greeter greeter = Greeter.newInstance();
    System.out.println(greeter.greet("Modular World"));
  }
}
```

Ниже приведена структура исходных файлов для обоих рассматриваемых здесь модулей.

```
com.horstmann.greet
module-info.java
com
  horstmann
    greet
      Greeter.java
      internal
         GreeterImpl.java

v2ch09.exportedpkg
module-info.java
  com
    horstmann
      hello
        HelloWorld.java
```

Чтобы построить данную прикладную программу, необходимо скомпилировать сначала модуль `com.horstmann.greet`, выполнив следующую команду:

```
javac com.horstmann.greet/module-info.java \
   com.horstmann.greet/com/horstmann/greet/Greeter.java \
   com.horstmann.greet/com/horstmann/greet \
      /internal/GreeterImpl.java
```

Затем следует скомпилировать модуль прикладной программы вместе с первым модулем по заданному пути к модулям:

```
javac -p com.horstmann.greet \
     v2ch09.exportedpkg/module-info.java \
     v2ch09.exportedpkg/com/horstmann/hello/HelloWorld.java
```

Наконец, остается лишь выполнить рассматриваемую здесь прикладную программу вместе с обоими модулями по заданному пути к модулям:

```
java -p v2ch09.exportedpkg:com.horstmann.greet \
     -m v2ch09.exportedpkg/com.horstmann.hello.HelloWorld
```

СОВЕТ. Чтобы построить ту же самую прикладную программу в интегрированной среде разработки Eclipse, придется создать в ней отдельный проект для каждого модуля, а затем отредактировать свойства проекта **`v2ch09.exportedpkg`**. Модуль **`com.horstmann.greet`** необходимо ввести в путь к модулям на вкладке Projects (рис. 9.2).

Как видите, операторы `requires` и `exports` образуют основание модульной системы на платформе Java. По существу, эта система довольно проста. В модулях объявляется, что конкретно им требуется и какие пакеты они предоставляют другим модулям. Незначительное отклонение от обычного назначения оператора `exports` будет представлено в разделе 9.12.

ВНИМАНИЕ! Модуль не обеспечивает область видимости. Следовательно, *не* допускается наличие пакетов с одинаковыми именами в *разных* модулях. И это справедливо даже для скрытых пакетов, которые не экспортируются.

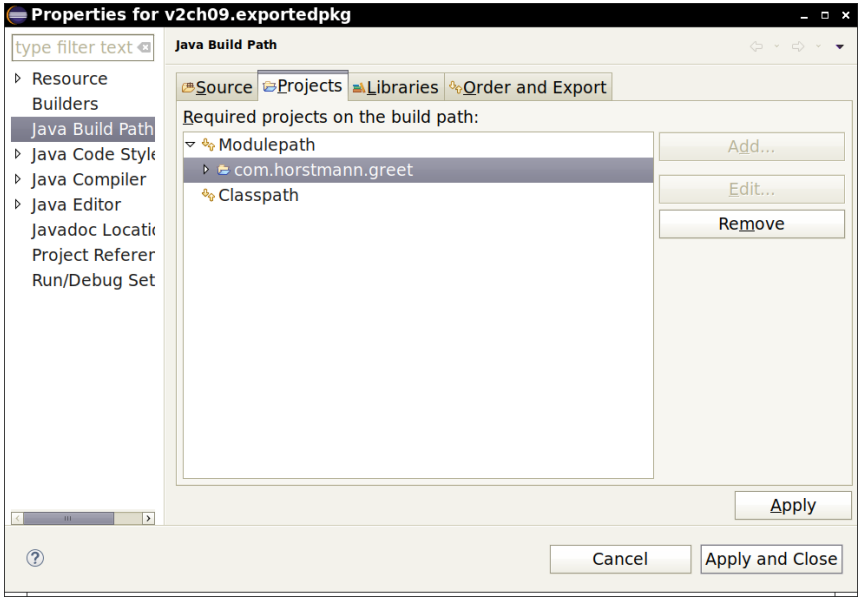


Рис. 9.2. Ввод зависимого модуля в проект, разрабатываемый в Eclipse

9.6. Модульные архивные JAR-файлы

До сих мы просто компилировали модули в дерево каталогов исходного кода. Очевидно, что это неудовлетворительно для развертывания модулей. Вместо этого для развертывания модулей достаточно разместить все его классы в архивном JAR-файле, а файл `module-info.class` — в корневом каталоге. Такой архивный JAR-файл называется *модульным*.

Чтобы создать модульный архивный JAR-файл, следует воспользоваться утилитой `jar`, как обычно. Если же имеется несколько пакетов, то компиляцию необходимо выполнить с параметром `-d`, в соответствии с которым файлы классов размещаются в отдельном каталоге, который создается, если он не существует. Затем можно выполнить утилиту `jar` с параметром `-C`, чтобы сменить данный каталог для накапливания файлов.

```
javac -d modules/com.horstmann.greet \
      $(find com.horstmann.greet -name *.java)
jar -cvf com.horstmann.greet.jar \
    -C modules/com.horstmann.greet .
```

Если для построения применяется такое инструментальное средство, как Maven, Ant или Gradle, то построение архивного JAR-файла следует выполнять, как обычно. И если в этот процесс включается файл `module-info.class`, то в конечном итоге получается модульный архивный JAR-файл. Этот архивный JAR-файл можно включить в путь к модулю, и тогда модуль будет загружен.

ВНИМАНИЕ! В прошлом классы из пакета иногда распространялись через целый ряд архивных JAR-файлов. (Такой пакет называется "разделенным".) Это, вероятно, была не сама удачная идея, а с появлением модулей она стала вообще неосуществимой.

Подобно обычным архивным JAR-файлам, в модульном архивном JAR-файле можно указать главный класс, как выделено ниже курсивом.

```
javac -p com.horstmann.greet.jar \
      -d modules/v2ch09.exportedpkg \
       $(find v2ch09.exportedpkg -name *.java)
jar -c -v -f v2ch09.exportedpkg.jar \
    -e com.horstmann.hello.HelloWorld \
    -C modules/v2ch09.exportedpkg .
```

Запуская прикладную программу на выполнение, следует указать модуль, содержащий главный класс:

```
java -p com.horstmann.greet.jar:v2ch09.exportedpkg.jar \
     -m v2ch09.exportedpkg
```

При создании архивного JAR-файла можно дополнительно указать номер версии. В частности, с помощью параметра `--module-version` и знака `@` имя архивного JAR-файла можно дополнить номером версии, как выделено ниже курсивом.

```
jar -c -v -f com.horstmann.greet@1.0.jar \
    --module-version 1.0 -C com.horstmann.greet .
```

Как обсуждалось ранее, номер версии не используется в модульной системе на платформе Java для разрешения модулей. Но он может быть запрошен другими инструментальными средствами и каркасами.

НА ЗАМЕТКУ! Выяснить номер версии можно средствами прикладного интерфейса API для рефлексии. Так, в следующем примере кода получается объект типа **Optional**, содержащий символьную строку **"1.0"** с искомым номером версии:

```
Optional<String> version = Greeter.class.getModule()
                           .getDescriptor().rawVersion();
```

НА ЗАМЕТКУ! Модульным эквивалентом загрузчика классов является *уровень*. В модульной системе на платформе Java модули из комплекта JDK и прикладных программ загружаются на *уровне начальной загрузки*. Прикладная программа может загрузить другие модули, используя прикладной интерфейс API уровней, который в данной книге не рассматривается. В такой программе могут приниматься во внимание версии модулей. При этом предполагается, что разработчики таких программ, как серверы приложений на платформе Java EE, будут пользоваться прикладным интерфейсом API уровней для поддержки модулей.

СОВЕТ. Если модуль требуется загрузить в командную оболочку консольной утилиты JShell, архивный JAR-файл следует включить в путь к модулю и указать параметр **--add-modules**, как показано ниже.

```
jshell --module-path com.horstmann.greet@1.0.jar \
       --add-modules com.horstmann.greet
```

9.7. Модули и рефлексивный доступ

Как было показано в предыдущих разделах, в модульной системе соблюдается инкапсуляция. В одном модуле могут быть доступны только те пакеты, которые экспортируются явным образом из другого модуля. В прошлом всегда имелась возможность преодолеть досадные ограничения на доступ с помощью рефлексии. Как пояснялось в главе 5 первого тома настоящего издания, рефлексия дает возможность получить доступ к закрытым членам любого класса.

Но в модульной системе такая возможность уже отсутствует. Если класс находится в модуле, то получить рефлексивный доступ к его неоткрытым членам не удастся. В качестве напоминания в приведенном ниже примере кода показано, получить доступ к закрытым членам класса.

```
Field f = obj.getClass().getDeclaredField("salary");
f.setAccessible(true);
double value = f.getDouble(obj);
f.setDouble(obj, value * 1.1);
```

Вызов `f.setAccessible(true)` завершится удачно, при условии, что диспетчер защиты не запретит доступ к закрытому полю. Но ведь прикладные программы на Java редко выполняются с диспетчерами защиты, а во многих библиотеках применяется рефлексивный доступ. Характерными тому примерами служат объектно-реляционные преобразователи вроде JPA, автоматически сохраняющие объекты в базах данных, а также библиотеки вроде JAXB или JSON-B, выполняющие взаимное преобразование объектов и данных формата XML или JSON.

Чтобы воспользоваться модулями из такой библиотеки, придется проявить особую осторожность. В качестве примера, демонстрирующего это положение, разместим класс `ObjectAnalyzer` из главы 5 первого тома настоящего издания в модуле `com.horstmann.util`. В этом классе имеется метод `toString()`, выводящий поля объекта с помощью рефлексии.

В отдельном модуле `v2ch09.openpkg` содержится приведенный ниже простой класс `Country`.

```
package com.horstmann.places;

public class Country
{
  private String name;
  private double area;

 public Country(String name, double area)
  {
    this.name = name;
    this.area = area;
  }
  // . . .
}
```

В следующей короткой программе демонстрируется порядок анализа объекта типа `Country`:

```
package com.horstmann.places;

import com.horstmann.util.*;
```

```
public class Demo
{
  public static void main(String[] args)
         throws ReflectiveOperationException
  {
    var belgium = new Country("Belgium", 30510);
    var analyzer = new ObjectAnalyzer();
    System.out.println(analyzer.toString(belgium));
  }
}
```

Теперь скомпилируем оба модуля и программу `Demo` следующим образом:

```
javac com.horstmann.util/module-info.java \
  com.horstmann.util/com/horstmann/util/ObjectAnalyzer.java
javac -p com.horstmann.util v2ch09.openpkg/module-info.java \
  v2ch09.openpkg/com/horstmann/places/*.java
java -p v2ch09.openpkg:com.horstmann.util \
     -m v2ch09.openpkg/com.horstmann.places.Demo
```

Выполнение данной программ завершится неудачно со следующим исключением:

```
Exception in thread "main"
  java.lang.reflect.InaccessibleObjectException:
Unable to make field private
  java.lang.String com.horstmann.places.Country.name
accessible: module v2ch09.openpkg does not
  "opens com.horstmann.places" to module com.horstmann.util[3]
```

Разумеется, чисто теоретически нельзя нарушать инкапсуляцию и манипулировать закрытыми членами объекта. Но такие механизмы, как XML-привязка или объектно-реляционное преобразование, настолько распространены, что модульной системе приходится приспосабливаться к ним.

Используя оператор `opens`, модуль может *открыть* пакет, обеспечивая тем самым доступ ко всем классам и их членам в данном пакете во время выполнения, а к закрытым членам — через рефлексию. Ниже показано, что для этого необходимо сделать. И благодаря таким изменениям класс `ObjectAnalyzer` будет действовать правильно.

```
module v2ch09.openpkg
{
  requires com.horstmann.util;
  opens com.horstmann.places;
}
```

Модуль можно объявить как открытый (`open`) следующим образом:

```
open module v2ch09.openpkg
{
```

---

[3] Исключение в "главном" потоке исполнения
java.lang.reflect.InaccessibleObjectException:
Не удалось получить доступ к полю
private java.lang.String com.horstmann.places.Country.name:
модуль v2ch09.openpkg не "открывает пакет com.horstmann.places"
для модуля java.xml.util

```
  requires com.horstmann.util;
}
```

Открытый модуль обеспечивает доступ во время выполнения ко всем его пакетам, как будто все они объявлены в операторах `exports` и `opens`. Но во время компиляции доступны только пакеты, экспортируемые явным образом. Открытые модули сочетают в себе безопасность модульной системы во время компиляции с классическим разрешительным поведением во время выполнения.

Как упоминалось в главе 5 первого тома настоящего издания, архивные JAR-файлы могут содержать, помимо файлов классов и манифестов, *файлы ресурсов*, которые можно загружать методом `Class.getResourceAsStream()`, а теперь и методом `Module.getResourceAsStream()`. Если ресурс хранится в каталоге, совпадающем с пакетом из модуля, то пакет должен стать открытым для вызывающего кода. Ресурсы из других каталогов, а также файлы классов и манифестов могут стать доступными для чтения кому угодно.

НА ЗАМЕТКУ! В качестве более практического примера объект типа **`Country`** можно было бы преобразовать в формат XML или JSON. Для преобразования в формат XML в состав версий Java 9 и 10 был включен модуль **`java.xml.bind`**, но затем он был исключен из состава версии Java 11 (вместе с модулями **`java.activation`**, **`java.corba`**, **`java.transaction`**, **`java.xml.ws`** и **`java.xml.ws.annotation`**). Эти модули содержат пакеты, являющиеся также частью спецификации Jakarta EE (ранее Java EE), где прикладные интерфейсы API являются более объемлющими, чем в Java EE. Корпоративные серверы приложений нельзя модуляризировать, если комплект JDK вступает в конфликт с пакетами. К сожалению, на момент написания данной книги модуляризированная замена для привязки данных формата XML отсутствовала.

Тем не менее в реализации JSON-B (стандартного уровня привязки данных в формате JSON) предоставляются модульные архивные JAR-файлы, если построить ее из исходного кода. Можно все же надеяться, что эти архивные JAR-файлы появятся в центральном хранилище Maven Central к тому времени, когда вы будете читать эти строки. В таком случае укажите эти архивные JAR-файлы в пути к модулям и выполните демонстрационную программу **`com.horstmann.places.Demo2`**. Преобразование в формат JSON завершится успешно, как только будет открыт пакет **`com.horstmann.places`**.

НА ЗАМЕТКУ! Вполне возможно, что в будущих библиотеках будут применяться *переменные дескрипторы* вместо рефлексии для чтения и записи данных в полях. Класс **`VarHandle`** подобен классу **`Field`**. С его помощью можно прочитать или записать данные в указанном поле любого экземпляра конкретного класса. Но для получения объекта типа **`VarHandle`** в коде библиотеки потребуется объект типа **`Lookup`**, как показано ниже.

```
public Object getFieldValue(Object obj, String fieldName,
                            Lookup lookup)
      throws NoSuchFieldException, IllegalAccessException
{
  Class<?> cl = obj.getClass();
  Field field = cl.getDeclaredField(fieldName);
  VarHandle handle = MethodHandles
                     .privateLookupIn(cl, lookup)
                     .unreflectVarHandle(field);
  return handle.get(obj);
}
```

Такой подход оказывается вполне работоспособным при условии, что объект типа **Lookup** формируется в том модуле, где имеется разрешение на доступ к полю. В каком-нибудь методе из модуля просто вызывается метод **MethodHandles.lookup()**, где выдается объект, инкапсулирующий права доступа для вызывающего кода. Подобным образом один модуль может дать разрешение на доступ к закрытым членам другого модуля. На практике необходимо решить, как дать такое разрешение с минимальными затратами сил и средств.

9.8. Автоматические модули

Итак, мы выяснили, как применять на практике модульную систему на платформе Java. Если начать с совершенно нового проекта, в котором весь код придется писать самостоятельно, то можно разработать модули, объявить зависимости от них и упаковать спроектированную прикладную программу в модульные архивные JAR-файлы.

Тем не менее это крайне редкий случай. Большинство разрабатываемых проектов опираются на сторонние библиотеки. Разумеется, можно подождать до тех пор, пока поставщики всех имеющихся библиотек не превратят их в модули, а затем модуляризировать свой код.

Но что делать, если ждать некогда? В модульной системе на платформе Java предоставляются два механизма для преодоления пропасти, образовавшейся между прикладными программами, разрабатывавшимися до появления модульной системы, и полностью модульными программами. Этими механизмами являются автоматические и безымянные модули.

В целях переноса прикладной программы любой архивный JAR-файл можно превратить в модуль, разместив его в каталоге по пути к модулю, а не к классу. Архивный JAR-файл без составляющей `module-info.class` в пути к модулю называется *автоматическим модулем*. Автоматический модуль подчиняется следующим правилам.

1. Модуль неявно содержит оператор `requires` для всех остальных модулей.
2. Все пакеты из данного модуля экспортируются и открываются.
3. Если в манифесте `META-INF/MANIFEST.MF` из архивного JAR-файла имеется запись с ключом `Automatic-Module-Name`, то его значением становится имя модуля.
4. В противном случае имя модуля получается из имени архивного JAR-файла, из которого исключается любой оконечный номер версии, а последовательности символов, не являющихся буквенно-цифровыми, заменяются точками.

В двух первых из перечисленных выше правил подразумевается, что пакеты в автоматическом модуле действуют так, как будто они указаны в пути к классам. Причина для применения пути к модулю заключается в выгоде, которую другие модули извлекают из выражения их зависимостей от данного модуля.

Допустим, требуется реализовать модуль, обрабатывающий файлы в формате CSV, применяя для этой цели библиотеку Apache Commons CSV, а в файле `module-info.java` — выразить зависимость данного модуля от библиотеки

Apache Commons CSV. Если ввести архивный файл `commons-csv-1.4.jar` в путь к модулю, то все модули прикладной программы смогут ссылаться на модуль данной библиотеки. Он называется `commons.csv` потому, что оконечный номер версии `-1.4` был удален из его имени, а символ, не являющийся буквенно-цифровым, заменен точкой.

Это имя может быть вполне приемлемым для обозначения модуля, поскольку библиотека Apache Commons CSV хорошо известна, и вряд ли кто-нибудь другой попробует воспользоваться тем же самым именем для обозначения другого модуля. Но было бы лучше, если бы те, кто сопровождает архивный JAR-файл этой библиотеки, охотно согласились обозначить его как модуль в обратном порядке следования доменных имен, желательно начиная с имени пакета верхнего уровня, как, например: `org.apache.commons.csv`. Для этого достаточно ввести следующую строку:

```
Automatic-Module-Name: org.apache.commons.csv
```

в файл манифеста `META-INF/MANIFEST.MF`, находящийся в архивном JAR-файле. Можно надеяться, что в конечном итоге они превратят свой архивный JAR-файл в настоящий модуль, введя файл `module-info.java` с зарезервированным именем модуля, чтобы все остальные модули, ссылающиеся на модуль библиотеки Apache Commons CSV, смогли и дальше работать в нормальном режиме.

НА ЗАМЕТКУ План перехода на модули является крупным социальным экспериментом, и никто не знает, чем он закончится. Следовательно, прежде чем указывать сторонние архивные JAR-файлы в пути к модулю, следует проверить, являются ли они модульными, а иначе — задано ли в их манифесте имя модуля. В противном случае архивный JAR-файл можно преобразовать в автоматический модуль, но с готовностью обновить имя модуля в дальнейшем.

На момент написания данной книги в версии 1.5 архивного JAR-файла библиотеки Apache Commons CSV отсутствовал дескриптор модуля или имя автоматического модуля. Тем не менее он будет действовать правильно по пути к модулям. Эту библиотеку можно загрузить по адресу `https://commons.apache.org/proper/commons-csv`, распаковать и разместить архивный файл `commons-csv-1.5.jar` в каталоге с модулем `v2ch9.automod`. Этот модуль содержит приведенную ниже простую программу, читающую данные о странах из файла формата CSV.

```
package com.horstmann.places;

import java.io.*;
import org.apache.commons.csv.*;

public class CSVDemo
{
  public static void main(String[] args) throws IOException
  {
    var in = new FileReader("countries.csv");
    Iterable<CSVRecord> records =
             CSVFormat.EXCEL.withDelimiter(';')
                      .withHeader().parse(in);
    for (CSVRecord record : records)
    {
```

```
      String name = record.get("Name");
      double area = Double.parseDouble(record.get("Area"));
      System.out.println(name + " has area " + area);
    }
  }
}
```

В данном случае архивный файл `commons-csv-1.5.jar` употребляется как автоматический модуль, поэтому его необходимо затребовать, как показано ниже.

```
@SuppressWarnings("module")
module v2ch09.automod
{
  requires commons.csv;
}
```

Ниже приведены команды для компиляции и выполнения данной программы.

```
javac -p v2ch09.automod:commons-csv-1.5.jar \
    v2ch09.automod/com/horstmann/places/CSVDemo.java \
    v2ch09.automod/module-info.java
java -p v2ch09.automod:commons-csv-1.5.jar \
     -m v2ch09.automod/com.horstmann.places.CSVDemo
```

9.9. Безымянные модули

Любой класс, отсутствующий в пути к модулю, является частью безымянного модуля. Формально может быть больше одного безымянного модуля, но все подобные модули должны действовать как единое целое, называемое *безымянным модулем*. Как и в автоматических модулях, в безымянном модуле могут быть доступны все остальные модули, а все его пакеты — экспортированы и открыты.

Тем не менее *ни в одном* из безымянных модулей недоступны другие безымянные модули. А *явным* называется такой модуль, который не является ни автоматическим, ни безымянным, т.е. это такой модуль, в пути которому указан файл `module-info.class`. Иными словами, явные модули никогда не приводят к путанице в путях к классам.

Рассмотрим в качестве примера программу из предыдущего раздела. Допустим, архивный файл `commons-csv-1.5.jar` требуется указать в пути к классам, а не к модулям, как показано ниже.

```
java --module-path v2ch09.automod \
     --class-path commons-csv-1.5.jar \
     -m v2ch09.automod/com.horstmann.places.CSVDemo
```

Теперь данная программа не запустится на выполнение, а вместо этого появится следующее сообщение об ошибке:

```
Error occurred during initialization of boot layer
java.lang.module.FindException: Module commons.csv
not found, required by v2ch09.automod[4]
```

[4] При инициализации уровня начальной загрузки возникла ошибка java.lang.module.FindException: модуль commons.csv, требующийся модулю v2ch09.automod, не найден

Таким образом, переход к модульной системе на платформе Java по необходимости оказывается восходящим процессом, как поясняется ниже.

1. Сама платформа Java модуляризирована.
2. Библиотеки модуляризированы с помощью автоматических модулей или же путем их преобразования в явные модули.
3. Как только все библиотеки, применяемые в прикладной программе, окажутся модуляризированными, ее исходный код можно перенести в модуль.

НА ЗАМЕТКУ! Из автоматических модулей *можно* прочитать безымянный модуль, а следовательно, их зависимости могут перейти в путь к классам.

9.10. Параметры командной строки для переноса прикладного кода

Даже если модули не применяются в прикладных программах, избежать модульной системы все равно не удастся при переходе к Java 9 и последующим версиям. И даже если прикладной код находится по пути к классам в безымянном модуле, а все пакеты экспортируются и открываются, то он все равно взаимодействует с платформой Java, которая модуляризирована.

Стандартное поведение до версии Java 11 состояло в том, чтобы разрешить недопустимый доступ к модулям, но при первой же попытке подобного нарушения вывести соответствующее предупреждение на консоль. В будущей версии Java такое стандартное поведение будет изменено, а недопустимый доступ отменен. Чтобы дать программирующим на Java время подготовиться к такому изменению, им предлагается проверять свои прикладные программы с помощью параметра `--illegal-access`. Ниже перечислены четыре возможных варианта его установки.

1. **`--illegal-access=permit`**. Задается стандартное поведение в версии Java 9, при первой попытке недопустимого доступа на консоль выводится предупреждающее сообщение.
2. **`--illegal-access=warn`**. При первой попытке недопустимого доступа на консоль выводится предупреждающее сообщение.
3. **`--illegal-access=debug`**. При первой попытке недопустимого доступа на консоль выводится предупреждающее сообщение и трассировка стека.
4. **`--illegal-access=deny`**. Задается будущее стандартное поведение, любой недопустимый доступ отменяется.

Теперь самое время проверить все сказанное выше на примере параметра `--illegal-access=deny`, чтобы подготовиться к тому моменту, когда такое поведение станет стандартным. Итак, рассмотрим прикладную программу, в которой применяется больше недоступный внутренний прикладной интерфейс API вроде `com.sun.rowset.CachedRowSetImpl`. Лучшим средством в данном случае будет смена реализации. (Начиная с версии Java 7 кешированный набор строк можно получить из таблицы с помощью класса `RowSetProvider`.) Но допустим,

что доступ к исходному коду отсутствует. В таком случае запустим прикладную программу с параметром `-add-exports`, указав модуль и пакет, который требуется экспортировать, а также модуль (в данном случае безымянный), куда следует экспортировать этот пакет.

```
java --illegal-access=deny --add-exports \
       java.sql.rowset/com.sun.rowset=ALL_UNNAMED \
      -jar MyApp.jar
```

А теперь допустим, что рефлексия применяется в прикладной программе для доступа к закрытым полям или методам. Рефлексия в безымянном модуле вполне допустима, но больше невозможна для рефлексивного доступа к неоткрытым членам классов на платформе Java. Например, в некоторых библиотеках, формирующих классы Java в динамическом режиме, защищенный метод `ClassLoader.defineClass()` вызывается через рефлексию. Если подобная библиотека применяется в прикладной программе, то в команду ее компиляции и запуска на выполнение следует ввести следующий параметр:

```
--add-opens java.base/java.lang=ALL-UNNAMED
```

Если ввести все эти параметры командной строки для обеспечения нормальной работоспособности унаследованного прикладного кода, то в конечном итоге получится кошмарная по своей сложности командная строка. Чтобы удобнее обращаться с этими параметрами, их можно ввести в один или несколько файлов с префиксом `@`, как демонстрируется в следующем примере:

```
java @options1 @options2 -jar MyProg.java
```

где файлы `options1` и `options2` содержат параметры команды `java`.

Для создания файлов параметров имеются следующие правила.

- Отдельные параметры разделяются знаками пробелов, табуляции или новой строки.
- Аргументы, включающие в себя пробелы (например, `"Program Files"`), заключаются в двойные кавычки.
- Строка, оканчивающаяся знаком `\`, соединяется со следующей строкой.
- Знаки обратной косой черты должны быть экранированы, как, например, `C:\\Users\\Fred`.
- Строки комментариев должны начинаться со знака `#`.

9.11. Переходные и статические требования

Основная форма оператора `requires` была рассмотрена в разделе 9.4. В этом разделе будут рассмотрены два варианта этого оператора, которые иногда оказываются полезными.

В некоторых случаях пользователю отдельного модуля может быть неудобно объявлять все требующиеся модули вручную. Рассмотрим в качестве примера модуль `javafx.controls`, содержащий такие элементы пользовательского интерфейса, построенного на основе библиотеки JavaFX, как экранные кнопки. Модуль `javafx.base` требуется модулю `javafx.controls` и всем, кто пользуется

модулем javafx.controls. (В отсутствие пакетов, доступных из модуля javafx.base, вряд ли удастся сделать что-нибудь с таким элементом управления из пользовательского интерфейса, как экранная кнопка типа Button.) Именно по этой причине требование модуля javafx.base объявляется в модуле javafx.controls с помощью модификатора transitive, как показано ниже. Теперь в любом модуле, где объявляется требование модуля javafx.controls, автоматически требуется и модуль javafx.base.

```
module javafx.controls {
  requires transitive javafx.base;
  . . .
}
```

НА ЗАМЕТКУ! Некоторые программисты рекомендуют всегда пользоваться оператором **requires transitive**, если пакет из другого модуля применяется в открытом прикладном интерфейсе API. Но эта рекомендация не подходит для программирования на языке Java. Рассмотрим в качестве примера следующий модуль:

```
module java.sql {
  requires transitive java.logging;
  . . .
}
```

Пакет из модуля **java.logging** применяется во всем прикладном интерфейсе API из модуля **java.sql** лишь один раз, а точнее, в методе **java.sql.Driver.parentLogger()**, возвращающем объект типа **java.util.logging.Logger**. Поэтому было бы вполне приемлемо не объявлять требование этого модуля как переходное. И тогда лишь в тех модулях, где применяется данный метод, необходимо объявить, что в них требуется модуль **java.logging**.

К числу наглядных примеров применения оператора requires transitive относится *агрегатный* модуль, в котором отсутствуют пакеты, а имеются только переходные требования. Таким является модуль java.se, который объявляется следующим образом:

```
module java.se {
  requires transitive java.compiler;
  requires transitive java.datatransfer;
  requires transitive java.desktop;
  . . .
  requires transitive java.sql;
  requires transitive java.sql.rowset;
  requires transitive java.xml;
  requires transitive java.xml.crypto;
}
```

Программисту, которого не интересуют мелкоструктурные зависимости от модулей, может просто потребоваться модуль java.se и доступ ко всем модулям на платформе Java SE.

Наконец, нечасто, но все же иногда применяется оператор requires static, в котором устанавливается, что модуль должен присутствовать во время компиляции, но совсем не обязательно во время выполнения. Этот оператор применяется в следующих случаях.

1. Для доступа к аннотации, которая обрабатывается во время компиляции и объявляется в другом модуле.
2. Для применения класса в другом модуле, если он доступен, а иначе — для выполнения каких-нибудь других действий, как, например, в следующем примере кода:

```
try {
  new oracle.jdbc.driver.OracleDriver();
  ...
} catch (NoClassDefFoundError er) {
  // выполнить что-нибудь другое
}
```

9.12. Уточненный экспорт и открытие модулей

В этом разделе рассматриваются варианты операторов `exports` и `opens`, сужающих их область видимости до указанного ряда модулей. Например, модуль `javafx.base` содержит следующий оператор:

```
exports com.sun.javafx.collections to javafx.controls,
        javafx.graphics, javafx.fxml, javafx.swing;
```

Такой оператор обозначает *уточненный экспорт*. Одни из перечисляемых в нем модулей могут иметь доступ к пакету, а другие — нет.

Чрезмерное употребление операторов уточненного экспорта может свидетельствовать о неудачно выбранной модульной структуре. Тем не менее потребность в них может возникнуть при модуляризации существующей кодовой базы. В данном случае разработчики платформы Java распределили код для JavaFX по нескольким модулям, и это трезвая мысль, поскольку не во всех JavaFX-приложениях требуется взаимодействие с FXML или Swing. Но реализаторы JavaFX свободно воспользовались в своем коде внутренними классами вроде `com.sun.javafx.collections.ListListenerHelper`. В проекте, разрабатываемом с нуля, вместо этого можно спроектировать более надежный открытый прикладной интерфейс API.

Аналогичным образом можно наложить ограничение на модули, указываемые в операторе `opens`. Например, в примерах кода из раздела 9.7 можно было бы употребить уточненный оператор `opens`, как выделено ниже полужирным. В итоге пакет `com.horstmann.places` будет открыт только для модуля `java.xml.util`.

9.13. Загрузка служб

Класс `ServiceLoader` (см. главу 6 первого тома настоящего издания) предоставляет облегченный механизм для согласования интерфейсов служб с их реализациями, а модульная система на платформе Java упрощает применение этого механизма.

Напомним вкратце, каким образом происходит загрузка службы. У службы имеется интерфейс и одна или несколько возможных ее реализаций. Ниже приведен простой пример интерфейса службы приветствий.

```
public interface GreeterService {
  String greet(String subject);
  Locale getLocale();
}
```

В одном или нескольких модулях предоставляются реализации данной службы, как, например, показано ниже.

```
public class FrenchGreeter implements GreeterService
{
  public String greet(String subject)
  { return "Bonjour " + subject; }
  public Locale getLocale()
  { return Locale.FRENCH; }
}
```

Потребитель службы должен выбрать одну из предоставляемых ее реализаций по наиболее подходящему для него критерию.

```
ServiceLoader<GreeterService> greeterLoader =
              ServiceLoader.load(GreeterService.class);
GreeterService chosenGreeter;
for (GreeterService greeter : greeterLoader) {
  if (...) {
  chosenGreeter = greeter;
  }
}
```

В прошлом реализации служб предоставлялись путем размещения текстовых файлов в каталоге `META-INF/services` из архивного JAR-файла, содержавшего классы реализаций, а ныне модульная система предлагает более совершенный способ. Вместо текстовых файлов в дескрипторы модулей вводятся соответствующие операторы.

В модуль, предоставляющий реализацию службы, вводится оператор `provides`, где перечисляется интерфейс этой службы, который может быть определен в любом модуле, а также реализующий ее класс, который должен быть составной часть данного модуля. В качестве примера ниже приведен фрагмент кода из модуля `jdk.security.auth`. По существу, это эквивалент текстового файла из каталога `META-INF/services`.

```
module jdk.security.auth {
  ...
  provides javax.security.auth.spi.LoginModule with
    com.sun.security.auth.module.Krb5LoginModule,
    com.sun.security.auth.module.UnixLoginModule,
    com.sun.security.auth.module.JndiLoginModule,
    com.sun.security.auth.module.KeyStoreLoginModule,
    com.sun.security.auth.module.LdapLoginModule,
    com.sun.security.auth.module.NTLoginModule;
}
```

В тех модулях, где употребляется данная служба, содержится оператор `uses`, как демонстрируется в следующем примере кода:

```
module java.base {
  . . .
  uses javax.security.auth.spi.LoginModule;
}
```

Когда в коде модуля, где используется служба, вызывается метод `ServiceLoader.load(`*интерфейс_службы*`.class)`, загружаются классы соответствующих поставщиков данной службы, даже если они могут и не оказаться в доступных пакетах.

В рассматриваемом здесь примере предоставляются реализации службы приветствия на немецком и французском языках в пакете `com.horstmann.greetsvc.internal`. Модуль этой службы экспортирует пакет `com.horstmann.greetsvc`, но не пакет с ее реализациями. В операторе `provides` сама служба и реализующие ее классы объявляются в неэкспортируемом пакете, как показано ниже.

```
module com.horstmann.greetsvc
{
  exports com.horstmann.greetsvc;

  provides com.horstmann.greetsvc.GreeterService with
    com.horstmann.greetsvc.internal.FrenchGreeter,
    com.horstmann.greetsvc.internal.GermanGreeterFactory;
}
```

Данная служба употребляется в модуле `v2ch09.useservice`. Пользуясь классом `ServiceLoader`, можно обойти все предоставляемые службы приветствия и выбрать среди них наиболее подходящую для конкретного языка, как демонстрируется в приведенном ниже фрагменте кода.

```
package com.horstmann.hello;

import java.util.*;
import com.horstmann.greetsvc.*;

public class HelloWorld
{
  public static void main(String[] args)
  {
    ServiceLoader<GreeterService> greeterLoader =
          ServiceLoader.load(GreeterService.class);
    String desiredLanguage =
                 args.length > 0 ? args[0] : "de";
    GreeterService chosenGreeter = null;
    for (GreeterService greeter : greeterLoader)
    {
      if (greeter.getLocale().getLanguage()
                .equals(desiredLanguage))
        chosenGreeter = greeter;
    }
    if (chosenGreeter == null)
      System.out.println("No suitable greeter.");
    else
      System.out.println(
                chosenGreeter.greet("Modular World"));
  }
}
```

В объявлении модуля требуется модуль службы и объявляется применение службы приветствия типа `GreeterService`, как показано ниже. В результате объявлений, сделанных в операторах `provides` и `uses`, модулю, употребляющему

указанную службу приветствия, разрешается доступ к открытым классам ее реализации.

```
module v2ch09.useservice
{
  requires com.horstmann.greetsvc;
  uses com.horstmann.greetsvc.GreeterService;
}
```

Чтобы построить и выполнить программу из данного примера, необходимо скомпилировать сначала службу, выполнив следующую команду:

```
javac com.horstmann.greetsvc/module-info.java \
      com.horstmann.greetsvc/com/horstmann/greetsvc \
         /GreeterService.java \
      com.horstmann.greetsvc/com/horstmann/greetsvc \
         /internal/*.java
```

Затем необходимо скомпилировать и выполнить модуль, употребляющий данную службу, выполнив команду

```
javac -p com.horstmann.greetsvc \
   v2ch09.useservice/com/horstmann/hello/HelloWorld.java \
   v2ch09.useservice/module-info.java
java -p com.horstmann.greetsvc:v2ch09.useservice \
     -m v2ch09.useservice/com.horstmann.hello.HelloWorld
```

9.14. Инструментальные средства для работы с модулями

Утилита `jdeps` служит для анализа зависимостей в отдельном ряде архивных JAR-файлов. Допустим, требуется модуляризировать инструментальное средство JUnit 4. Для выявления в нем зависимостей необходимо выполнить следующую команду:

```
jdeps -s junit-4.12.jar hamcrest-core-1.3.jar
```

По заданному параметру `-s` выводится следующий итоговый результат анализа зависимостей:

```
hamcrest-core-1.3.jar -> java.base
junit-4.12.jar -> hamcrest-core-1.3.jar
junit-4.12.jar -> java.base
junit-4.12.jar -> java.management
```

По этому результату можно построить приведенный на рис. 9.3 граф модулей.

Если опустить параметр `-s`, то после сводки модулей будет выведено соответствие одних пакетов и требующихся для них других пакетов и модулей. Если же добавить параметр `-v`, то будет выведено соответствие классов и требующихся для них других пакетов и модулей. По заданному параметру `--generate-module-info` выводятся файлы `module-info` для каждого проанализированного модуля:

```
jdeps --generate-module-info /tmp/junit junit-4.12.jar \
        hamcrest-core-1.3.jar
```

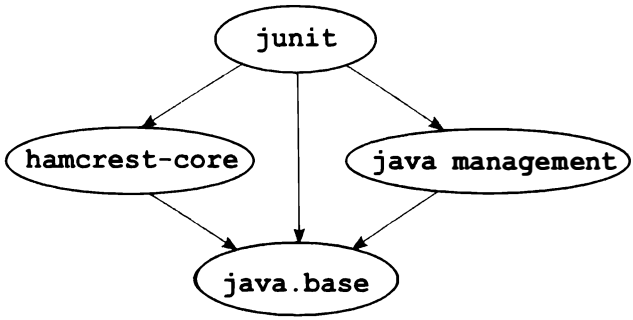


Рис. 9.3. Граф модулей, построенный утилитой **jdeps**

НА ЗАМЕТКУ! Имеется также параметр для формирования графического вывода на "точечном" языке описания графов. Если установлено инструментальное средство **dot**, то, выполнив следующие команды:

```
jdeps -s -dotoutput /tmp/junit junit-4.12.jar \
      hamcrest-core-1.3.jar
dot -Tpng /tmp/junit/summary.dot > /tmp/junit/summary.png
```

можно получить файл изображения **summary.png** с приведенным на рис. 9.4 графом модулей.

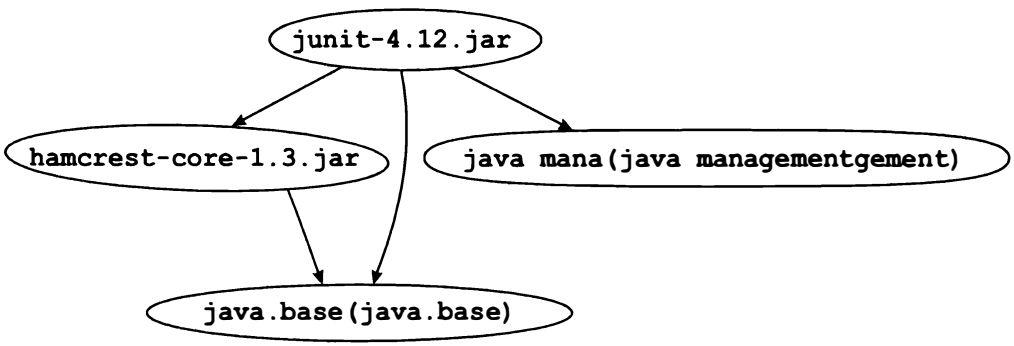


Рис. 9.4. Файл изображения **summary.png**

Утилита `jlink` служит для построения прикладной программы, которая выполняется без отдельной исполняющей среды Java. В итоге получается намного более компактный образ, чем целый комплект JDK. При этом указываются модули, которые требуется включить, а также выходной каталог:

```
jlink --module-path com.horstmann.greet.jar: \
        v2ch09.exportedpkg.jar:$JAVA_HOME/jmods \
      --add-modules v2ch09.exportedpkg --output /tmp/hello
```

В выходном каталоге находится подкаталог `bin` с исполняемым файлом `java`. Так, если выполнить следующую команду:

```
bin/java -m v2ch09.exportedpkg
```

из главного класса модуля будет вызван метод `main()`.

Назначение утилиты `jlink` состоит в том, чтобы скомплектовать минимальный ряд модулей, которые требуются для выполнения прикладной программы. Все эти модули можно перечислить по следующей команде:

```
bin/java --list-modules
```

В рассматриваемом здесь примере выводится такой результат:

```
v2ch09.exportedpkg
com.horstmann.greet
java.base@9
```

Все модули включаются в файл `lib/modules` *образа на стадии выполнения*. На моем компьютере этот файл занимает 23 Мбайт, тогда как образ всех модулей JDK на стадии выполнения — 181 Мбайт, а вся прикладная программа — 45 Мбайт, т.е. на 10% меньше, чем весь комплект JDK, занимающий 486 Мбайт.

Данное инструментальное средство может послужить основанием для упаковки прикладной программы. Тем не менее придется получить ряд файлов для разных платформ и запустить на выполнение соответствующие сценарии для данной прикладной программы.

НА ЗАМЕТКУ! Проверить образ на стадии выполнения можно по команде **`jimage`**. Тем не менее образы на стадии выполнения формируются во внутреннем для виртуальной машины Java формате и не предназначены для применения в других инструментальных средствах.

Наконец, утилита `jmod` служит для построения и проверки файлов модулей, включаемых в состав комплекта JDK. Если обратиться к каталогу `jmods` в комплекте JDK, то в нем можно обнаружить файл с расширением `jmod` для каждого включенного в данный комплект модуля. Но в нем больше нет файла `rt.jar`.

Подобно архивным JAR-файлам, эти файлы содержат файлы классов. Они могут также содержать библиотеки платформенно-ориентированного кода, команды, файлы заголовков и конфигурации, а также правовые положения. В JMOD-файлах применяется формат ZIP, поэтому их содержимое можно проверить средствами ZIP.

В отличие от архивных JAR-файлов, JMOD-файлы полезны только для компоновки, т.е. получения образов на стадии выполнения. Потребность в получении JMOD-файлов возникает лишь в том случае, если требуется скомплектовать вместе с модулями двоичные файлы, например, для библиотек платформенно-ориентированного кода.

НА ЗАМЕТКУ! Файлы **`rt.jar`** и **`tools.jar`** больше не входят в состав версии Java 9, поэтому необходимо обновить любые ссылки на них. Так, если в файле правил защиты делается ссылка на файл **`tools.jar`**, эту ссылку необходимо заменить ссылкой на соответствующий модуль:

```
grant codeBase "jrt:/jdk.compiler" {
  permission java.security.AllPermission;
};
```

Синтаксис **jrt:** обозначает файл времени выполнения Java.

На этом глава, посвященная модульной системе на платформе Java, завершается. В следующей главе рассматривается еще одна важная тема безопасности. Безопасность всегда была отличительной особенностью платформы Java. В связи с постоянным увеличением степени риска в мире, в котором мы живем, ясное представление о средствах безопасности Java становится все более актуальным для многих разработчиков.

ГЛАВА 10

Безопасность

В этой главе...

- Загрузчики классов
- Диспетчеры защиты и полномочия
- Аутентификация пользователей
- Цифровые подписи
- Шифрование

С появлением технологии Java специалисты оценили по достоинству не только хорошо продуманные выразительные средства этого языка, но и механизмы обеспечения безопасности при выполнении аплетов, доставляемых через Интернет. Очевидно, что доставка исполняемых аплетов имеет смысл только тогда, когда получатели уверены, что код не может нанести ущерба работе их компьютеров. Поэтому вопросы безопасности были и остаются основной заботой как разработчиков, так и пользователей технологии Java. Это означает, что, в отличие от других языков и систем, в которых безопасность обеспечивалась в последнюю очередь, механизмы защиты изначально стали неотъемлемой частью технологии Java.

В технологии Java безопасность обеспечивают следующие три механизма.

- Структурные функциональные возможности языка (например, проверка границ массивов, запрет на преобразования непроверенных типов данных, отсутствие указателей и т.д.).
- Средства контроля доступа, определяющие действия, которые разрешается или запрещается выполнять в коде (например, может ли код получать доступ к файлам, передавать данные по сети и т.д.).
- Механизм цифровой подписи, предоставляющий авторам возможность применять стандартные алгоритмы для аутентификации своих программ, а пользователям — точно определять, кто создал код и изменился ли он с момента его подписания.

В этой главе сначала рассматриваются *загрузчики классов*, проверяющие файлы классов на предмет целостности при их загрузке в виртуальную машину Java. А затем будет показано, каким образом этот механизм может выявлять в файлах классов признаки злонамеренных действий.

Для обеспечения максимальной безопасности оба механизма загрузки классов (используемый по умолчанию и специальный) должны взаимодействовать с классом *диспетчера защиты*, определяющим действия, которые разрешено или запрещено выполнять в коде. Поэтому далее в этой главе подробно поясняется, каким образом настраивается безопасность на платформе Java.

И в завершение главы будут рассмотрены различные алгоритмы шифрования, доступные в пакете `java.security` и позволяющие подписывать код и обеспечивать аутентификацию пользователей. Как обычно, основное внимание здесь уделяется только тем вопросам, которые представляют наибольший интерес для разработчиков прикладных программ на Java. А тем, кому требуется более углубленное изучение данной темы, рекомендуем книгу *Inside Java 2 Platform Security: Architecture, API Design, and Implementation, 2nd edition* Ли Гонга, Гэри Эллисона и Мэри Дейджфорд (Li Gong, Gary Ellison, Mary Dageforde; издательство Prentice Hall PTR, 2003 г.).

10.1. Загрузчики классов

Компилятор Java преобразует исходные операторы языка в понятный для виртуальной машины Java байт-код, который сохраняется в файле класса с расширением `.class`. В каждом файле класса содержится код определения и реализации только для одного класса или интерфейса. В последующих разделах будет показано, каким образом виртуальная машина Java загружает файлы классов.

10.1.1. Процесс загрузки классов

Виртуальная машина Java загружает только те файлы классов, которые требуются для выполнения программы в данный момент. Допустим, выполнение программы начинается с файла `MyProgram.class`. Ниже описаны действия, которые выполняет виртуальная машина Java.

1. В виртуальной машине имеется механизм загрузки файлов классов, например, путем их чтения с диска или загрузки из Интернета. С помощью этого механизма сначала загружается содержимое файла класса `MyProgram`.
2. Если в классе `MyProgram` встречаются поля или объекты, ссылающиеся на классы других типов, то дополнительно загружаются файлы этих классов. (Процесс загрузки всех классов, от которых зависит данный класс, называется *разрешением класса*.)
3. Затем виртуальная машина выполняет метод `main()` из класса `MyProgram`. (Этот метод является статическим, поэтому никаких экземпляров класса `MyProgram` создавать не требуется.)
4. Если для выполнения метода `main()` или вызываемого из него метода требуются дополнительные классы, они загружаются далее из соответствующих файлов.

Но в механизме загрузки классов используется не один, а несколько загрузчиков. Каждой программе на Java сопутствует по меньшей мере три загрузчика классов:

- загрузчик базовых классов;
- загрузчик платформных классов;
- загрузчик системных классов, иногда называемый загрузчиком прикладных классов.

Загрузчик базовых классов загружает платформные классы, содержащиеся в следующих модулях:

```
java.base
java.datatransfer
java.desktop
java.instrument
java.logging
java.management
java.management.rmi
java.naming
java.prefs
java.rmi
java.security.sasl
java.xml
```

а также целый ряд внутренних модулей из комплекта JDK.

Загрузчику базовых классов не соответствует ни один из объектов типа `ClassLoader`. Например, при вызове следующего метода возвращается пустое значение `null`:

```
String.class.getClassLoader()
```

До версии Java 9 классы платформы Java находились в архивном файле `rt.jar`. А поскольку платформа Java стала теперь модульной, каждый ее модуль содержится в JMOD-файле (см. главу 9). Загрузчик платформных классов загружает все классы платформы Java, которые были еще загружены загрузчиком базовых классов. Наконец, загрузчик системных классов загружает все прикладные классы по пути к модулям или классам.

НА ЗАМЕТКУ! До версии Java 9 загрузчик расширений классов загружал стандартные расширения из каталога **`jre/lib/ext`**, а механизм замены утвержденных норм обеспечивал порядок замены некоторых платформных классов, включая реализации CORBA и XML, их новыми версиями. Оба эти механизма исключены из текущей версии Java.

10.1.2. Иерархия загрузчиков классов

Загрузчики классов связаны отношениями "родитель–потомок". У каждого загрузчика классов, за исключением базовых классов, имеется свой родительский загрузчик классов. Предполагается, что загрузчик классов дает возможность своему родителю загружать любой нужный класс и загружает его сам только в том случае, если этого не может сделать родитель. Так, если загрузчик системных классов получает запрос на загрузку системного класса (например, `java.lang.`

StringBuilder), сначала он предлагает загрузить его загрузчику платформных классов. Загрузчик платформных классов, в свою очередь, предлагает сделать это загрузчику базовых классов. В итоге загрузчик базовых классов находит и загружает класс из архивного файла rt.jar, а все остальные загрузчики классов прекращают дальнейший поиск.

Некоторые программы имеют модульную архитектуру, где определенные части кода упаковываются в дополнительные, но необязательно подключаемые модули. Если подключаемые модули упаковываются в архивные JAR-файлы, то классы этих модулей могут загружаться с помощью экземпляра класса URLClassLoader, как показано ниже.

```
var url = new URL("file:///path/to/plugin.jar");
var pluginLoader = new URLClassLoader(new URL[] { url });
Class<?> cl = pluginLoader.loadClass("mypackage.MyClass");
```

В конструкторе класса URLClassLoader вообще не указан родитель загрузчика подключаемых модулей типа pluginLoader, поэтому в роли его родителя будет выступать загрузчик системных классов. Схематическое представление иерархии загрузчиков классов приведено на рис. 10.1.

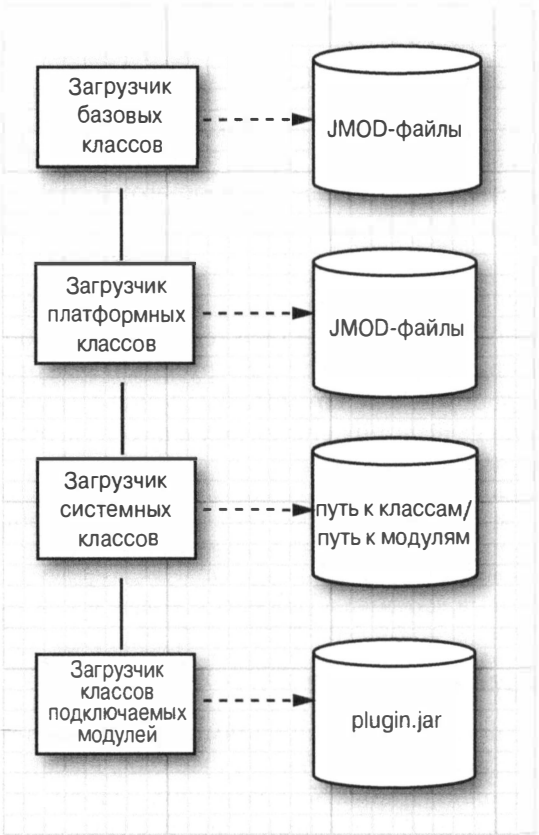


Рис. 10.1. Иерархия загрузчиков классов

ВНИМАНИЕ! До версии Java 9 загрузчик системных классов был представлен экземпляром класса **URLClassLoader**. Некоторые программисты обычно прибегали к приведению типов, чтобы получить доступ к методу **getURLs()**, или же вводили архивные JAR-файлы в путь к классам, вызывая защищенный метод **addURLs()** через рефлексию. Но теперь и то и другое стало невозможным.

Как правило, разработчиков прикладных программ мало интересует иерархия загрузчиков классов. Обычно одни классы загружаются потому, что этого требуют другие классы, и этот процесс является абсолютно прозрачным для разработчика. Но иногда разработчикам прикладных программ все-таки приходится вмешиваться в данный процесс, чтобы указывать нужный загрузчик классов. Рассмотрим следующий пример.

- В коде прикладной программы имеется вспомогательный метод, вызывающий метод `Class.forName(classNameString)`.
- Этот метод вызывается из класса подключаемого модуля.
- Параметр `classNameString` указывает на класс, содержащийся в архивном JAR-файле подключаемого модуля.

Автор подключаемого модуля имеет все основания предполагать, что его класс будет загружаться. Но загрузка класса вспомогательного метода выполнялась загрузчиком системных классов, т.е. именно тем загрузчиком, который использовался методом `Class.forName()`. Это означает, что классы, находящиеся в архивном файле `plugin.jar`, недоступны. Подобное явление называется *инверсией загрузчиков классов*.

Для устранения данного препятствия требуется, чтобы вспомогательный метод использовал нужный загрузчик классов. А для этого можно указать нужный загрузчик классов в виде параметра или установить его в качестве загрузчика контекста классов в текущем потоке исполнения. Вторая методика применяется во многих каркасах (например, JAXP и JNDI).

В каждом потоке исполнения имеется ссылка на загрузчик классов, называемый *загрузчиком контекста классов*. Таким загрузчиком для главного потока исполнения является загрузчик системных классов. При создании нового потока исполнения для него назначается загрузчик контекста классов из того потока исполнения, который его создает. Следовательно, если не вмешиваться в этот процесс, то для всех потоков исполнения в качестве загрузчика контекста классов назначается загрузчик системных классов. Тем не менее существует возможность указать любой другой загрузчик классов, сделав следующий вызов:

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

Вспомогательный метод может затем извлечь загрузчик контекста классов следующим образом:

```
Thread t = Thread.currentThread();
ClassLoader loader = t.getContextClassLoader();
Class cl = loader.loadClass(className);
```

СОВЕТ. При написании метода, загружающего класс по имени, вызывающему его коду рекомендуется предоставлять возможность выбирать между передачей явно задаваемого загрузчика классов и применением загрузчика контекста классов. Просто использовать загрузчик класса этого метода нежелательно.

10.1.3. Применение загрузчиков классов в качестве пространств имен

Рис. 10.2. Два загрузчика классов загружают разные классы с одинаковыми именами

Всякому программирующему на Java известно, что для устранения конфликтов имен применяются имена пакетов. Например, в стандартной библиотеке Java имеются два класса под именем `Date`, но полностью они именуются как `java.util.Date` и `java.sql.Date`. Короткие имена употребляются для удобства программирования, но требуют включения соответствующих операторов `import` в исходный код. В выполняющейся программе все имена классов содержат имена своих пакетов.

Но, как ни странно, в пределах одной и той же виртуальной машины могут существовать два разных класса, имеющих одинаковое имя класса и пакета. Дело в том, что класс определяется по его полному имени и загрузчику класса. Такая технология очень удобна при загрузке кода из нескольких источников. Например, на сервере приложений для загрузки каждого приложения используются отдельные загрузчики классов. Это позволяет виртуальной машине различать классы из разных приложений независимо от их имен. Допустим, на сервере приложений загружаются два разных приложения, и в каждом из них имеется класс `Util`. Но поскольку каждый класс загружается отдельным загрузчиком классов, эти классы полностью различаются и не конфликтуют между собой (рис. 10.2).

10.1.4. Создание собственного загрузчика классов

Для специальных целей можно создавать и свои собственные загрузчики классов. Такой подход позволяет выполнять какие-нибудь специальные проверки, прежде чем передавать байт-код виртуальной машине. Например, можно создать

загрузчик классов, способный запрещать загрузку класса, который не был помечен как оплаченный. Чтобы создать собственный загрузчик классов, достаточно расширить класс `ClassLoader` и переопределить метод `findClass(String className)`.

Метод `loadClass()` из суперкласса `ClassLoader` берет на себя все обязанности по делегированию функций загрузки родительскому загрузчику классов и вызывает метод `findClass()` только в том случае, если класс еще не был загружен и если родительскому загрузчику классов не удалось его загрузить. Поэтому при самостоятельной реализации метода `findClass()` необходимо сделать так, чтобы он выполнял следующее.

1. Загружал байт-код класса из локальной файловой системы или какого-нибудь другого источника.
2. Вызывал метод `defineClass()` из суперкласса `ClassLoader` для представления байт-кода виртуальной машине.

В примере программы из листинга 10.1 демонстрируется реализация загрузчика классов, способного загружать файлы зашифрованных классов. Эта программа сначала запрашивает у пользователя имя первого загружаемого класса (т.е. класса, содержащего метод `main()`) и ключ расшифровки. Затем она использует специальный загрузчик классов для загрузки указанного класса и вызова метода `main()`. Этот специальный загрузчик расшифровывает указанный класс и все несистемные классы, на которые он ссылается, после чего программа наконец-то вызывает метод `main()` из загруженного класса (рис. 10.3). Ради простоты для шифрования файлов классов в рассматриваемом здесь примере используется древний шифр Юлия Цезаря, пренебрегая более чем 2000-летним опытом в области криптографии.

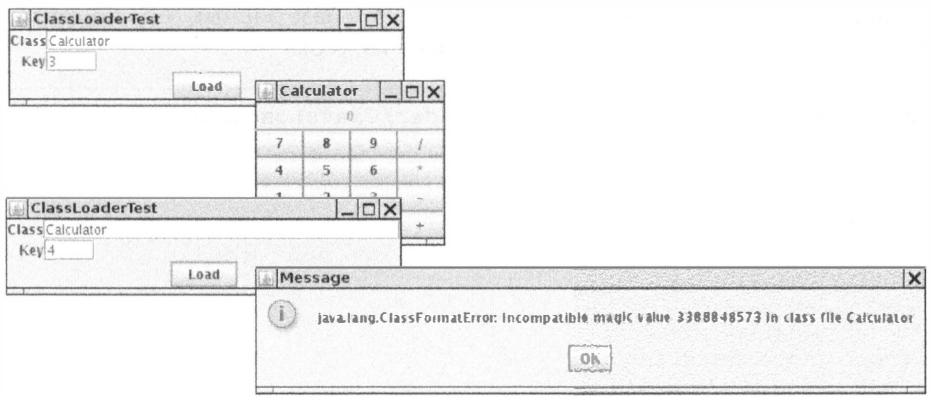


Рис. 10.3. Программа **`ClassLoaderTest`**

НА ЗАМЕТКУ! В прекрасной книге *The Codebreakers* Дэвида Кана (David Kahn; издательство Macmillan, 1967 г.[1]) говорится, что Юлий Цезарь кодировал 24 буквы латинского алфавита, сдвигая их на 3 буквы, чем повергал в недоумение всех своих противников. На момент

[1] В русском переводе эта книга вышла под названием *Взломщики кодов* в издательстве "Центрполиграф", М. 2000 г.

написания первоначального варианта этой главы правительство США ограничивало экспорт наиболее сложных методов шифрования. Поэтому использование довольно простого метода шифрования Юлия Цезаря в рассматриваемом здесь примере никак не нарушит экспортных ограничений.

В рассматриваемом здесь варианте тайнописи Юлия Цезаря в качестве ключа используется число в пределах от 1 до 255. Для шифрования байт суммируется с этим ключом по модулю 256. Алгоритм шифрования реализован в программе из файла `Caesar.java`, исходный код которой приведен в листинге 10.2. Чтобы не поставить в тупик стандартный загрузчик классов, для зашифрованных файлов классов используется расширение `.caesar`.

При расшифровке загрузчик классов вычитает значение ключа из каждого байта. В прилагаемом к данной книге коде предлагаются четыре файла классов, зашифрованных с помощью традиционного значения 3 ключа шифрования. Эти классы нельзя загрузить и запустить на выполнение в стандартной виртуальной машине Java. Для этого понадобится специальный загрузчик классов, реализованный в рассматриваемой здесь программе `ClassLoaderTest`.

Зашифрованные файлы классов имеют самое разное практическое применение (если, конечно, используется более сложный шифр, чем тайнопись Юлия Цезаря). Без ключа шифрования эти файлы классов бесполезны, поскольку они не могут выполняться в стандартной виртуальной машине Java, а восстановить их исходный код нелегко.

Это означает, что для аутентификации пользователя класса или для проверки того, что программа оплачена до запуска, можно воспользоваться специальным загрузчиком классов. Естественно, что шифрование является только одним из возможных применений специального загрузчика классов. Для решения других задач, например, сохранения файлов классов в базе данных, можно создавать и применять иные разновидности загрузчиков классов.

Листинг 10.1. Исходный код из файла **`classLoader/ClassLoaderTest.java`**

```
1  package classLoader;
2
3  import java.io.*;
4  import java.lang.reflect.*;
5  import java.nio.file.*;
6  import java.awt.*;
7  import java.awt.event.*;
8  import javax.swing.*;
9
10 /**
11 * В этой программе демонстрируется специальный загрузчик
12  * классов, расшифровывающий файлы классов
13  * @version 1.25 2018-05-01
14  * @author Cay Horstmann
15  */
16 public class ClassLoaderTest
17 {
18   public static void main(String[] args)
```

```
19    {
20      EventQueue.invokeLater(() ->
21        {
22          var frame = new ClassLoaderFrame();
23          frame.setTitle("ClassLoaderTest");
24          frame.setDefaultCloseOperation(
25                JFrame.EXIT_ON_CLOSE);
26          frame.setVisible(true);
27        });
28    }
29  }
30
31  /**
32   * Этот фрейм содержит два текстовых поля для ввода
33   * имени загружаемого класса и ключ расшифровки
34   *
35   */
36  class ClassLoaderFrame extends JFrame
37  {
38    private JTextField keyField =
39            new JTextField("3", 4);
40    private JTextField nameField =
41            new JTextField("Calculator", 30);
42    private static final int DEFAULT_WIDTH = 300;
43    private static final int DEFAULT_HEIGHT = 200;
44
45    public ClassLoaderFrame()
46    {
47      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
48      setLayout(new GridBagLayout());
49      add(new JLabel("Class"),
50          new GBC(0, 0).setAnchor(GBC.EAST));
51      add(nameField, new GBC(1, 0).setWeight(100, 0)
52                                  .setAnchor(GBC.WEST));
53      add(new JLabel("Key"),
54          new GBC(0, 1).setAnchor(GBC.EAST));
55      add(keyField, new GBC(1, 1).setWeight(100, 0)
56                                 .setAnchor(GBC.WEST));
57      var loadButton = new JButton("Load");
58      add(loadButton, new GBC(0, 2, 2, 1));
59      loadButton.addActionListener(event ->
60                          runClass(nameField.getText(),
61                          keyField.getText()));
62      pack();
63    }
64
65    /**
66     * Выполняет главный метод указанного класса
67     * @param name имя класса
68     * @param key Ключ расшифровки файлов классов
69     */
70    public void runClass(String name, String key)
71    {
72      try
73      {
```

```
74        var loader = new CryptoClassLoader(
75                        Integer.parseInt(key));
76        Class<?> c = loader.loadClass(name);
77        Method m = c.getMethod("main", String[].class);
78        m.invoke(null, (Object) new String[] {});
79      }
80      catch (Throwable t)
81      {
82        JOptionPane.showMessageDialog(this, t);
83      }
84    }
85  }
86
87  /**
88   * Этот загрузчик классов загружает их
89   * из зашифрованных файлов
90   */
91  class CryptoClassLoader extends ClassLoader
92  {
93    private int key;
94
95    /**
96     * Конструирует загрузчик зашифрованных классов
97     * @param k клюя расшифровки
98     */
99    public CryptoClassLoader(int k)
100   {
101     key = k;
102   }
103
104   protected Class<?> findClass(String name)
105             throws ClassNotFoundException
106   {
107     try
108     {
109       byte[] classBytes = null;
110       classBytes = loadClassBytes(name);
111       Class<?> cl = defineClass(name, classBytes, 0,
112                                 classBytes.length);
113       if (cl == null)
114           throw new ClassNotFoundException(name);
115       return cl;
116     }
117     catch (IOException e)
118     {
119       throw new ClassNotFoundException(name);
120     }
121   }
122
123   /**
124    * Загружает и расшифровывает байты из файла класса
125    * @param name Имя класса
126    * @return Массив с байтами из файла класса
127    */
128   private byte[] loadClassBytes(String name)
```

```
129           throws IOException
130    {
131      String cname = name.replace('.', '/') + ".caesar";
132      byte[] bytes = Files.readAllBytes(Paths.get(cname));
133      for (int i = 0; i < bytes.length; i++)
134        bytes[i] = (byte) (bytes[i] - key);
135      return bytes;
136    }
137 }
```

Листинг 10.2. Исходный код из файла **classLoader/Caesar.java**

```
1  package classLoader;
2
3  import java.io.*;
4
5  /**
6   * Шифрует файл, используя тайнопись Юлия Цезаря
7   * @version 1.02 2018-05-01
8   * @author Cay Horstmann
9   */
10 public class Caesar
11 {
12   public static void main(String[] args) throws Exception
13   {
14     if (args.length != 3)
15     {
16       System.out.println("USAGE: java "
17                 + " classLoader.Caesar in out key");
18       return;
19     }
20
21     try (var in = new FileInputStream(args[0]);
22          var out = new FileOutputStream(args[1]))
23     {
24       int key = Integer.parseInt(args[2]);
25       int ch;
26       while ((ch = in.read()) != -1)
27       {
28         byte c = (byte) (ch + key);
29         out.write(c);
30       }
31     }
32   }
33 }
```

java.lang.Class 1.0

- **ClassLoader getClassLoader()**
  Получает загрузчик для загрузки данного класса.

java.lang.ClassLoader 1.0

- **ClassLoader getParent()** 1.2

  Получает родительский загрузчик классов или пустое значение **null**, если это загрузчик базовых классов.

- **static ClassLoader getSystemClassLoader()** 1.2

  Получает загрузчик системных классов, т.е. тот, который применялся для загрузки первого прикладного класса.

- **protected Class findClass(String *name*)** 1.2

  Должен быть переопределен в загрузчике классов для поиска байт-кода класса и его представления виртуальной машине благодаря вызову метода **defineClass()**. Для отделения имени пакета от имени класса следует использовать точку и не указывать суффикс **.class**.

- **Class defineClass(String *name*, byte[] *byteCodeData*, int *offset*, int *length*)**

  Передает виртуальной машине новый класс, предоставляя байт-код в определенном диапазоне данных.

java.net.URLClassLoader 1.2

- **URLClassLoader(URL[] *urls*)**
- **URLClassLoader(URL[] *urls*, ClassLoader *parent*)**

  Создают загрузчик классов по указанному URL. Если URL оканчивается знаком **/**, то подразумевается каталог, а иначе — архивный JAR-файл.

java.lang.Thread 1.0

- **ClassLoader getContextClassLoader()** 1.2

  Получает загрузчик классов, обозначенный создателем данного потока исполнения как наиболее приемлемый для использования в этом потоке.

- **void setContextClassLoader(ClassLoader *loader*)** 1.2

  Устанавливает загрузчик классов для кода в данном потоке исполнения. Если при запуске потока на исполнение загрузчик контекста классов не задан явно, то используется родительский загрузчик контекста классов.

10.1.5. Верификация байт-кода

Когда загрузчик классов представляет виртуальной машине байт-код класса, этот код сначала обследуется *верификатором*. Верификатор проверяет все классы за исключением системных и определяет те команды, которые могут нанести ущерб виртуальной машине.

Ниже приведены некоторые виды проверок, выполняемых верификатором.

- Инициализация переменных перед их использованием.
- Согласование типов ссылок при вызове метода.

- Соблюдение правил доступа к закрытым данным и методам.
- Доступ к локальным переменным в стеке во время выполнения.
- Отсутствие переполнения стека.

При невыполнении какой-нибудь из этих проверок класс считается поврежденным и загружаться не будет.

НА ЗАМЕТКУ! Теорема Гёделя утверждает, что невозможно разработать алгоритмы, способные обработать программу и выяснить, обладает ли она конкретным свойством (например, независимостью от переполнения стека). В таком случае возникает вопрос: каким образом верификатор может определить, что в файле класса не соблюдается соответствие типов, используются неинициализированные переменные и переполняется стек? Нет ли здесь противоречия законам логики? Никакого противоречия нет, потому что верификатор *не* является алгоритмом принятия решений в смысле теоремы Гёделя. Если верификатор принимает программу, значит, она действительно безопасна. Но он может и отклонять команды виртуальной машины, даже если те выглядят вполне безопасными. (Вам, вероятно, уже доводилось сталкиваться с подобным затруднением и инициализировать переменную каким-нибудь фиктивным значением из-за того, что компилятор не в состоянии гарантировать ее должную инициализацию.)

Такая строгая верификация требуется по соображениям безопасности. Случайные ошибки, связанные с использованием неинициализированных переменных, могут легко нанести значительный ущерб программе, если вовремя не определить их. Еще важнее то обстоятельство, что при повсеместном доступе к Интернету необходимо обеспечить надежную защиту от злонамеренных попыток недобросовестных программистов, стремящихся проникнуть в чужую систему или нанести ей ущерб. Например, видоизменив значения в стеке выполняемой программы или в закрытых полях системных объектов, любая программа способна нарушить систему защиты веб-браузера.

Зачем же в таком случае создавать специальный верификатор для проверки всех этих условий? Ведь компилятор все равно не позволит сгенерировать файл класса, в котором предполагается использовать неинициализированную переменную или область закрытых данных, доступную из другого класса. В самом деле, файл класса, сформированный компилятором Java, всегда проходит верификацию. Но используемый в этих файлах формат байт-кода хорошо документирован, и всякий, имеющий опыт программирования на ассемблере и работы в шестнадцатеричном редакторе, может без труда создать вручную файл класса с достоверными, но потенциально опасными командами для виртуальной машины Java. Таким образом, верификатор блокирует злонамеренно измененные файлы классов, а не только проверяет созданные компилятором файлы.

В листинге 10.3 приведен пример программы `VerifierTest`, демонстрирующий изменение файла класса вручную. В этой простой программе вызывается метод `fun()` и выводится результат его выполнения. Она может быть запущена из командной строки или в виде аплета. Метод `fun()` выполняет сложение чисел `1` и `2`, как показано ниже.

```
static int fun()
{
  int m;
  int n;
```

```
  m = 1;
  n = 2;
  int r = m + n;
  return r;
}
```

В качестве эксперимента попробуйте скомпилировать приведенный ниже видоизмененный вариант метода `fun()`.

```
static int fun()
{
  int m = 1;
  int n;
  m = 1;
  m = 2;
  int r = m + n;
  return r;
}
```

В данном случае переменная **n** не инициализирована и может принимать произвольное значение. Естественно, что компилятор обнаружит эту ошибку и откажется компилировать программу. Для создания недопустимого файла класса придется немного потрудиться. Сначала запустите утилиту **javap**, чтобы выяснить, каким образом компилятор транслирует метод `fun()`:

```
javap -c verifier.VerifierTest
```

Эта утилита показывает байт-код из файла класса в мнемоническом виде:

```
Method int fun()
  0 iconst_1
  1 istore_0
  2 iconst_2
  3 istore_1
  4 iload_0
  5 iload_1
  6 iadd
  7 istore_2
  8 iload_2
  9 ireturn
```

Чтобы изменить команду `3`, т.е. заменить `istore_1` на `istore_0`, воспользуйтесь шестнадцатеричным редактором. Таким образом, локальная переменная `0` (т.е. `m`) будет инициализирована дважды, а локальная переменная `1` (т.е. `n`) вообще не будет инициализирована. Чтобы сделать это, необходимо знать шестнадцатеричные значения команд. Эти значения приводятся ниже и взяты из книги *The Java Virtual Machine Specification* Тима Линдхольма и Фрэнка Йеллина (Tim Lindholm & Frank Yellin; издательство Prentice Hall PTR, 1999 г.).

```
0 iconst_1 04
1 istore_0 3B
2 iconst_2 05
3 istore_1 3C
4 iload_0  1A
5 iload_1  1B
6 iadd     60
7 istore_2 3D
```

```
8 iload_2  1C
9 ireturn  AC
```

Отредактировать любое их этих значений можно в шестнадцатеричном редакторе. На рис. 10.4 приведено рабочее окно шестнадцатеричного редактора Gnome с загруженным файлом `VerifierTest.class`, в котором светло-серым выделены байт-коды метода `fun()`.

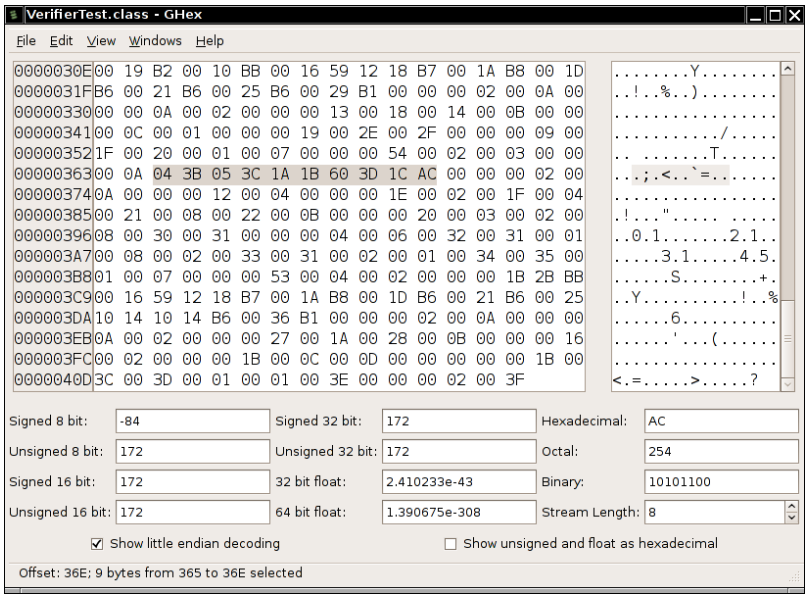


Рис. 10.4. Видоизменение байт-кода в шестнадцатеричном редакторе

Замените шестнадцатеричное значение `3C` на `3B`, сохраните файл, а затем попытайтесь запустить программу `VerifierTest` на выполнение. В итоге вы получите следующее сообщение об ошибке:

```
Exception in thread "main" java.lang.VerifyError:
    (class: VerifierTest, method:fun signature: ()I)
     Accessing value from uninitialized register 1[2]
```

Таким образом, виртуальная машина Java обнаружила недопустимое видоизменение байт-кода. Теперь запустите программу с параметром `-noverify` (или `-Xverify:none`), т.е. без верификации, как показано ниже.

Метод `fun()` возвращает случайную сумму, получаемую в результате сложения числа 2 с произвольным значением, хранящимся в переменной `n`, которая не была инициализирована. Ниже приведен типичный результат выполнения рассматриваемой здесь программы без верификации байт-кода.

```
1 + 2 == 15102330
```

---

[2] Исключение в "главном" потоке исполнения java.lang.VerifyError:
    (класс: VerifierTest, метод: method:fun signature: ()I)
     Доступ к значению из неинициализированного регистра 1

Листинг 10.3. Исходный код из файла **verifier/VerifierTest.java**

```
1  package verifier;
2
3  import java.awt.*;
4
5  /**
6   * этой прикладной программе демонстрируется
7   * верификатор байт-кода виртуальной машины Java.
8   * Если воспользоваться шестнадцатеричным редактором
9   * для видоизменения файла класса, то виртуальная
10  * машина Java должна обнаружить злонамеренное
11  * искажение содержимого файла класса
12  * @version 1.10 2018-05-05
13  * @author Cay Horstmann
14  */
15 public class VerifierTest
16 {
17   public static void main(String[] args)
18   {
19     System.out.println("1 + 2 == " + fun());
20   }
21
22   /**
23    * функция, вычисляющая сумму чисел 1 + 2
24    * @return Сумму 3, если код не нарушен
25    */
26   public static int fun()
27   {
28     int m;
29     int n;
30     m = 1;
31     n = 2;
32     // воспользоваться шестнадцатеричным редактором,
33     // чтобы изменить значение переменной m на 2
34     // в файле класса
35     int r = m + n;
36     return r;
37   }
38 }
```

10.2. Диспетчеры защиты и полномочия

После загрузки класса в виртуальную машину Java и проверки верификатором в действие вступает второй механизм обеспечения безопасности на платформе Java: диспетчер защиты. Этой теме посвящены последующие разделы.

10.2.1. Проверка полномочий

Диспетчер защиты проверяет, разрешено ли прикладному коду выполнять ту или иную операцию. Ниже перечислены операции, подвергаемые проверке в диспетчере защиты. Существует немало других проверок, выполняемых диспетчером защиты в библиотеке Java.

- Создание нового загрузчика классов.
- Выход из виртуальной машины.
- Получение доступа к члену другого класса с помощью рефлексии.
- Получение доступа к файлу.
- Подключение через сетевой сокет.
- Запуск задания на печать.
- Получение доступа к системному буферу обмена.
- Получение доступа к очереди событий в AWT.
- Обращение к окну верхнего уровня.

Поведение по умолчанию при запуске прикладных программ на Java *не* предусматривает никакой установки диспетчера защиты, поэтому все перечисленные выше операции оказываются разрешенными. И напротив, аплеты опираются на сильно ограничивающие правила защиты. Более строгая защита целесообразна и в других случаях.

Допустим, вы выполняете в своей системе экземпляр контейнера сервлетов Tomcat и разрешаете сотрудникам или учащимся устанавливать сервлеты. Но вам бы не хотелось, чтобы любой из них вызывал метод `System.exit()`, поскольку это прекращало бы действие экземпляра Tomcat. В таком случае вы можете установить такие правила защиты, которые приводят к генерированию исключения системы защиты при вызове метода `System.exit()` вместо прекращения работы виртуальной машины. В частности, метод `exit()` из класса `Runtime` вызывает метод `checkExit()` диспетчера защиты. Ниже приведен исходный код метода `exit()`.

```
public void exit(int status)
{
  SecurityManager security = System.getSecurityManager();
  if (security != null)
    security.checkExit(status);
  exitInternal(status);
}
```

Диспетчер защиты проверяет, откуда поступил запрос на завершение работы виртуальной машины: из браузера или из отдельного сервлета. Если диспетчер защиты дает разрешение на выполнение этого запроса, метод `checkExit()` возвращает управление и далее процесс обработки продолжается обычным образом. Но если диспетчер защиты не запрещает выполнение запроса, то метод `checkExit()` генерирует исключение типа `SecurityException`.

Метод `exit()` продолжает выполняться только в том случае, если никакого исключения не было. Затем он вызывает *закрытый платформенно-ориентированный* метод `exitInternal()`, который и завершает работу виртуальной машины Java. Другого способа для завершения работы виртуальной машины Java не существует, а поскольку метод `exitInternal()` закрытый, то он не может вызываться из какого-нибудь другого класса. Следовательно, всякий код, делающий попытку выйти из виртуальной машины Java, должен вызывать метод `exit()`, а следовательно, проходить проверку защиты, выполняемую методом `checkExit()`, но без генерирования соответствующего исключения.

Очевидно, что целостность правил защиты зависит от тщательности программирования. Поставщики системных служб в стандартной библиотеке должны всегда обращаться к диспетчеру защиты перед попытками выполнить любые серьезные операции.

Диспетчер защиты на платформе Java позволяет как программистам, так и системным администраторам организовать тщательно продуманное управление отдельными полномочиями. Более подробно об этом речь пойдет в следующем разделе, где сначала приводится краткий обзор модели защиты, реализованной на платформе Java 2, а затем показывается, как организуется управление правами доступа с помощью *файлов правил защиты*, и, наконец, объясняется, как определять свои собственные виды прав доступа.

10.2.2. Организация защиты на платформе Java

Модель защиты в JDK версии 1.0 была очень проста: локальные классы имели все полномочия, а возможности удаленных классов были ограничены так называемой "песочницей", т.е. диспетчер защиты аплетов отказывал в любом доступе к локальным ресурсам. Подобно тому, как ребенку разрешается играть в песочнице, строго ограничивая его действия ее пределами, удаленные классы имели возможность лишь выводить сведения на экран и взаимодействовать с пользователем. В версии JDK 1.1 эта модель была немного видоизменена, т.е. удаленный код, на который был подписан доверенный объект, получал те же права доступа, что и локальные классы. Тем не менее обе версии модели защиты были организованы по принципу "все или ничего", т.е. программы получали полный доступ к ресурсам, или же их действие ограничивалось пределами "песочницы".

Начиная с версии Java SE 1.2 на платформе Java предоставляется намного более гибкий механизм защиты. *Правила защиты* преобразуют *источники кода* в *наборы полномочий*, как показано на рис. 10.5.

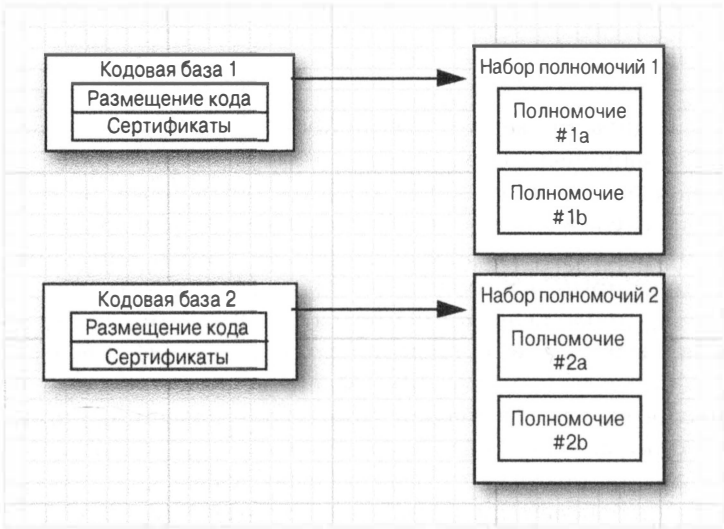


Рис. 10.5. Правила защиты

Каждый *источник кода* характеризуется *кодовой базой* и *набором сертификатов*. Кодовая база указывает место происхождения кода. Например, кодовой базой кода удаленного аплета является URL типа HTTP, откуда этот аплет загружен, а кодовой базой кода в архивном JAR-файле — URL, по которому загружен этот файл. Сертификаты, если таковые имеются, предоставляются соответствующей организацией и служат гарантией того, что данный код не является подделкой. Подробнее о сертификатах речь пойдет далее в этой главе.

Под *полномочиями* (иначе — правами доступа или привилегиями) подразумеваются любые свойства, которые проверяются диспетчером защиты. На платформе Java поддерживается целый ряд представляющих полномочия классов, каждый из которых инкапсулирует подробности конкретных полномочий. В качестве примера ниже приведено получение экземпляра класса `FilePermission`, дающего разрешение на чтение и запись любого файла в каталоге `/tmp`.

```
var p = new FilePermission("/tmp/*", "read,write");
```

Но еще важнее, что в стандартной реализации класс `Policy` считывает полномочия из *файла прав доступа*. В этом файле те же самые полномочия на чтение и запись, что и выше, выражаются следующим образом:

```
permission java.io.FilePermission "/tmp*", "read,write";
```

Более подробно файлы прав доступа рассматриваются в следующем разделе. На рис. 10.6 представлена иерархия классов полномочий, которые предоставлялись в версии Java 1.2. В последующих версиях Java было внедрено немало других классов полномочий.

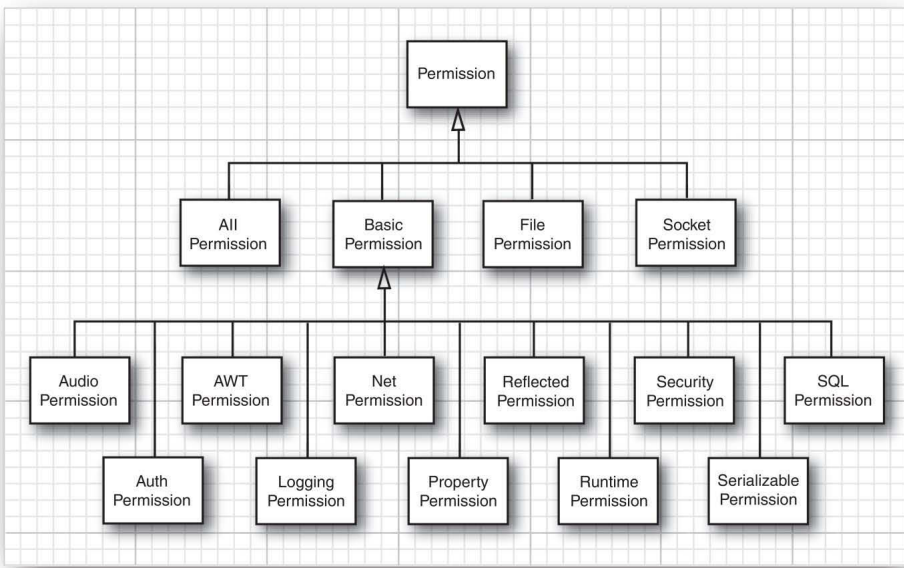


Рис. 10.6. Часть существующей иерархии классов полномочий

Как было показано в предыдущем разделе, в классе `SecurityManager` имеются методы проверки защиты типа `checkExit()`. Эти методы существуют только

для удобства программирования и обратной совместимости. Все они преобразуются в стандартные проверки полномочий. В качестве примера ниже приведен исходный код метода `checkExit()`.

```
public void checkExit()
{
  checkPermission(new RuntimePermission("exitVM"));
}
```

У каждого класса имеется *домен защиты*, т.е. объект, который инкапсулирует как источник кода, так и набор прав доступа этого класса. При необходимости проверить то или иное полномочие диспетчер защиты типа `SecurityManager` сначала выясняет, к каким классам относятся все методы, находящиеся в настоящий момент в стеке вызовов. Затем он получает доступ к доменам защиты всех этих классов и выясняет, допускает ли их набор полномочий выполнение операции, проверяемой в данный момент. Если все домены дают разрешение на выполнение данной операции, то проверка завершается успешно, в противном случае генерируется исключение типа `SecurityException`.

А почему разрешение на выполнение операции должны давать все методы в стеке вызовов? Для ответа на этот вопрос обратимся к конкретному примеру. Допустим, что в методе `init()` из сервлета требуется открыть файл с помощью следующего вызова:

```
var in = new FileReader(name);
```

Конструктор класса `FileReader` вызывает конструктор класса `FileInputStream`, который, в свою очередь, обращается к методу `checkRead()` диспетчера защиты, а тот вызывает метод `checkPermission()` с объектом, получаемым из конструктора `FilePermission(name, "read")`. Внешний вид стека вызовов в этом случае приведен в табл. 10.1.

Таблица 10.1. Стек вызовов во время проверки полномочий

Класс	Метод	Источник кода	Полномочия
SecurityManager	**checkPermission**	**null**	**AllPermission**
SecurityManager	**checkRead**	**null**	**AllPermission**
FileInputStream	**Constructor**	**null**	**AllPermission**
FileReader	**Constructor**	**null**	**AllPermission**
Сервлет	**init**	Источник кода сервлета	Полномочия веб-приложений в Tomcat

Классы `FileInputStream` и `SecurityManager` являются системными, их источник кода — пустым значением `null`, а набор полномочий — экземпляром класса `AllPermission`, который разрешает выполнять все операции. Очевидно, что полномочия только этих классов не могут определить исход проверки. Помимо них, метод `checkPermission()` должен принять во внимание и ограниченные полномочия класса сервлета. Благодаря проверке всего стека вызовов механизм защиты исключает вероятность выполнения одним классом важных операций от имени какого-нибудь другого класса.

НА ЗАМЕТКУ! Приведенное выше краткое описание проверки полномочий дает лишь самое общее представление о данном процессе. Многие технические детали в этом описании были опущены. Учитывая тот факт, что в вопросах безопасности вся суть скрывается именно в деталях, для ознакомления с ними рекомендуется книга *Securing Java: Getting Down to Business with Mobile Code* Гэри Макгроу и Эда Фельтена (Gary McGraw & Ed Felten; издательство Wiley, 1999 г.). Электронная версия книги оперативно доступна по адресу **www.securingjava.com**.

java.lang.SecurityManager 1.0

- **void checkPermission(Permission *p*)** 1.2

  Проверяет, предоставляет ли диспетчер защиты указанные полномочия. Если не предоставляет, то генерируется исключение типа **SecurityException**.

java.lang.Class 1.0

- **ProtectionDomain getProtectionDomain()** 1.2

  Получает доступ к представляющему данный класс домену защиты или возвращает пустое значение **null**, если этот класс загружен без домена защиты.

java.security.ProtectionDomain 1.2

- **ProtectionDomain(CodeSource *source*, PermissionCollection *permissions*)**

  Создает домен защиты, исходя из указанного источника кода и прав доступа.

- **CodeSource getCodeSource()**

  Получает источник кода из данного домена защиты.

- **boolean implies(Permission *p*)**

  Возвращает логическое значение **true**, если указанные полномочия разрешены в данном домене защиты.

java.security.CodeSource 1.2

- **Certificate[] getCertificates()**

  Получает цепочку сертификатов для подписей файлов классов, связанных с данным источником кода.

- **URL getLocation()**

  Получает кодовую базу файлов классов, связанных с данным источником кода.

10.2.3. Файлы правил защиты

Диспетчер правил защиты считывает содержимое *файлов правил защиты*, т.е. инструкции для преобразования источников кода в полномочия. Ниже приведен пример типичного файла правил защиты. Этот файл предоставляет полномочия на чтение и запись файлов в каталог `/tmp` любому коду, загруженному по адресу `http://www.horstmann.com/classes`.

```
grant codeBase "http://www.horstmann.com/classes"
{
  permission java.io.FilePermission "/tmp/*", "read,write";
};
```

Устанавливаться файлы правил защиты могут только в стандартных местах. По умолчанию такими местами являются:

- файл `java.policy` в основном каталоге платформы Java;
- файл `.java.policy` в рабочем каталоге пользователя (обратите внимание на точку, стоящую в начале его имени).

НА ЗАМЕТКУ! Стандартные пути к этим файлам можно изменить в конфигурационном файле **`java.security`**, находящемся в каталоге ***`jre/lib/security`***. По умолчанию пути к этим файлам указываются в конфигурационном файле и выглядят следующим образом:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

Системный администратор может редактировать файл **`java.security`** и указывать для правил защиты URL, находящиеся на каком-нибудь другом сервере, исключая всякую возможность для пользователей вносить изменения в эти правила. В файле правил защиты можно указать сколько угодно URL (по порядку следования номеров). Полномочия во всех подобных файлах все равно объединяются.

Если требуется, чтобы правила защиты хранились вне файловой системы, можно сначала реализовать подкласс, производный от класса **`Policy`** и собирающий сведения обо всех полномочиях, а затем изменить строку в конфигурационном файле **`java.security`** следующим образом:

```
policy.provider=sun.security.provider.PolicyFile
```

Во время тестирования не имеет смысла постоянно вносить изменения в стандартные файлы правил защиты, поэтому рекомендуется разместить сначала все полномочия в отдельном файле, например, в файле `MyApp.policy`, а затем указать явным образом имя этого файла для каждой прикладной программы. Применить правила защиты можно двумя способами. Во-первых, установить системное свойство в главном методе прикладной программы:

```
System.setProperty("java.security.policy", "MyApp.policy");
```

Во-вторых, запустить виртуальную машину, введя следующую команду:

```
java -Djava.security.policy=MyApp.policy MyApp
```

В приведенных выше примерах файл `MyApp.policy` добавляется к остальным действующим правилам защиты. Но если добавить еще один знак равенства, т.е. ввести следующую команду:

```
java -Djava.security.policy==MyApp.policy MyApp
```

то в прикладной программе будет использоваться *только* указанный файл правил защиты, а все стандартные файлы правил защиты — игнорироваться.

ВНИМАНИЕ! Во время тестирования можно легко допустить ошибку, случайно оставив в текущем каталоге файл **.java.policy**, который может предоставлять многие, а то и все возможные полномочия (**AllPermission**). Поэтому, заметив, что приложение как будто не обращает внимания на ограничения, указанные в добавленном файле правил защиты, следует прежде всего проверить, не остался ли в текущем каталоге файл **.java.policy**. Вероятность допустить подобную ошибку наиболее велика на системах UNIX, где файлы, имена которых начинаются с точки, по умолчанию не отображаются.

Как упоминалось ранее, диспетчер защиты в прикладных программах на Java по умолчанию не устанавливается. Это означает, что до тех пор, пока он не будет установлен, обнаружить действие файлов правил защиты не удастся. Установить диспетчер защиты можно двумя способами. Во-первых, добавить в тело метода `main()` следующую строку кода:

```
System.setSecurityManager(new SecurityManager());
```

Во-вторых, указать параметр `-Djava.security.manager` при запуске виртуальной машины Java из командной строки, как показано ниже.

```
java -Djava.security.manager
     -Djava.security.policy=MyApp.policy MyApp
```

Далее в этом разделе подробно рассматриваются способы описания полномочий в файле правил защиты и его формата, кроме способов указания сертификатов, речь о которых пойдет ниже. Итак, в файле правил защиты содержится последовательность записей `grant`, каждая из которых имеет следующий вид:

```
grant источник_кода
{
  полномочие1;
  полномочие2;
  ...
}
```

В источнике кода содержатся сведения о кодовой базе (которая может опускаться, если данная запись `grant` распространяется на код из всех источников), а также имена доверенных уполномоченных и подписавших сертификаты лиц (которые могут опускаться, если подписи не являются обязательными для данной записи). Сведения о кодовой базе указываются следующим образом:

```
codeBase "url"
```

Наличие косой черты (/) в конце URL означает, что ссылка делается на каталог, а отсутствие этого знака — на архивный JAR-файл, как показано в приведенных ниже примерах.

```
grant codeBase "www.horstmann.com/classes/" { . . . };
grant codeBase "www.horstmann.com/classes/MyApp.jar"
      { . . . };
```

В качестве разделителя файлов в URL кодовой базы должны всегда использоваться знаки косой черты, даже если речь идет об URL со ссылками на файлы в операционной системе Windows, как в следующем примере:

```
grant codeBase "file:C:/myapps/classes/" { . . . };
```

НА ЗАМЕТКУ! Как известно, URL типа **http** всегда начинаются двумя знаками косой черты (**http://**). Но в отношении URL типа **file**, по-видимому, возникает путаница из-за того, что средство чтения файлов правил защиты разрешает использовать для них два формата: **file://*локальный_файл*** и **file:*локальный_файл***. Более того, последняя косая черта перед именем диска в Windows не является обязательной. Это означает, что приемлемыми считаются все следующие варианты:

```
file:C:/dir/filename.ext
file:/C:/dir/filename.ext
file://C:/dir/filename.ext
file:///C:/dir/filename.ext
```

Как показывают проведенные нами проверки, на самом деле вариант **file:////C:/dir/filename.ext** также оказывается приемлемым, но найти этому разумное объяснение нам так и не удалось.

НА ЗАМЕТКУ! Рассмотрим в качестве примера приложение, компилирующее некоторый код Java и требующее для этой цели немало полномочий. До версии JDK 9 все полномочия можно было предоставить прикладному коду в архивном файле **tools.jar**. Но этого архивного JAR-файла больше не существует, поэтому требующиеся полномочия должны быть предоставлены соответствующему модулю, как показано ниже.

```
grant codeBase "jrt:/jdk.compiler"
{
  permission java.security.AllPermission;
};
```

Полномочия задаются в следующей синтаксической форме:

```
permission ИмяКласса ИмяЦелевогоОбъекта, СписокДействий;
```

Вместо параметра *ИмяКласса* указывается полностью уточненное имя класса, представляющего конкретные полномочия (например, `java.io.FilePermission`), а вместо параметра *ИмяЦелевогоОбъекта* — конкретный целевой объект, на который должно распространяться действие указываемых полномочий. Так, для полномочий доступа к файлам это может быть имя файла и каталога, а для полномочий доступа к сетевым сокетам — название и номер порта хоста. Наконец, вместо параметра *СписокДействий* указывается перечень охватываемых данным полномочием допустимых действий (например, чтения или установления соединения), разделяемых запятой. В некоторых классах полномочий не требуется указывать ни имена целевых объектов, ни списки действий. В табл. 10.2 перечислены некоторые из наиболее употребительных классов полномочий и охватываемые ими действия.

Таблица 10.2. Классы полномочий и связанные с ними целевые объекты и действия

Полномочия	Целевые объекты	Действия
java.io.FilePermission	Файлы (см. описание в тексте)	**read**, **write**, **execute**, **delete**
java.net.SocketPermission	Сокеты (см. описание в тексте)	**accept**, **connect**, **listen**, **resolve**
java.util.PropertyPermission	Свойства (см. описание в тексте)	**read**, **write**

Продолжение табл. 10.2

Полномочия	Целевые объекты	Действия
`java.lang.RuntimePermission`	`createClassLoader`, `getClassLoader`, `setContextClassLoader`, `enableContextClassLoaderOverride`, `createSecurityManager`, `setSecurityManager`, `exitVM`, `getenv.variableName`, `shutdownHooks`, `setFactory`, `setIO`, `modifyThread`, `stopThread`, `modifyThreadGroup`, `getProtectionDomain`, `readFileDescriptor`, `writeFileDescriptor`, `loadLibrary.libraryName`, `accessClassInPackage.packageName`, `defineClassInPackage.packageName`, `accessDeclaredMembers.className`, `queuePrintJob`, `getStackTrace`, `setDefaultUncaughtExceptionHandler`, `preferences`, `usePolicy`	Отсутствуют
`java.awt.AWTPermission`	`showWindowWithoutWarningBanner`, `accessClipboard`, `accessEventQueue`, `createRobot`, `fullScreenExclusive`, `listenToAllAWTEvents`, `readDisplayPixels`, `replaceKeyboardFocusManager`, `watchMousePointer`, `setWindowAlwaysOnTop`, `setAppletStub`	Отсутствуют
`java.net.NetPermission`	`setDefaultAuthenticator`, `specifyStreamHandler`, `requestPasswordAuthentication`, `setProxySelector`, `getProxySelector`, `setCookieHandler`, `getCookieHandler`, `setResponseCache`, `getResponseCache`	Отсутствуют
`java.lang.reflect.ReflectPermission`	`suppressAccessChecks`	Отсутствуют
`java.io.SerializablePermission`	`enableSubclassImlementation`, `enableSubstitution`	Отсутствуют
`java.security.SecurityPermission`	`createAccessControlContext`, `getDomainCombiner`, `getPolicy`, `setPolicy`, `getProperty.keyName`, `setProperty.keyName`, `insertProvider.providerName`, `removeProvider.providerName`, `setSystemScope`, `setIdentityPublicKey`, `setIdentityInfo`, `addIdentityCertificate`, `removeIdentityCertificate`, `printIdentity`, `clearProviderProperties.providerName`, `putProviderProperty.providerName`, `removeProviderProperty.providerName`, `getSignerPrivateKey`, `setSignerKeyPair`	Отсутствуют

Окончание табл. 10.2

Полномочия	Целевые объекты	Действия
java.security.AllPermission	Отсутствуют	Отсутствуют
javax.audio.AudioPermission	Воспроизведение записи	Отсутствуют
javax.security.auth.AuthPermission	**doAs**, **doAsPrivileged**, **getSubject**, **getSubjectFromDomainCombiner**, **setReadOnly**, **modifyPrincipals**, **modifyPublicCredentials**, **modifyPrivateCredentials**, **refreshCredential**, **destroyCredential**, **createLoginContext.contextName**, **getLoginConfiguration**, **setLoginConfiguration**, **refreshLoginConfiguration**	Отсутствуют
java.util.logging.LoggingPermission	**control**	Отсутствуют
java.sql.SQLPermission	**setLog**	Отсутствуют

Как следует из табл. 10.2, большинство полномочий просто позволяют выполнять определенные операции. К операции можно относиться как к целевому объекту с подразумеваемым действием "permit" (разрешить). Все классы полномочий, перечисленные в табл. 10.2, являются производными от класса BasicPermission (см. рис. 10.6). Но классы с такими целевыми объектами, как файл, сетевой сокет и свойство, оказываются более сложными и поэтому заслуживают дополнительного рассмотрения.

Целевые объекты прав доступа к файлам могут иметь следующую форму:

файл	Файл
каталог/	Каталог
каталог/*	Все файлы из данного каталога
*	Все файлы из текущего каталога
каталог/-	Все файлы из данного каталога и всех его подкаталогов
-	Все файлы из текущего каталога и всех его подкаталогов
<<ALL FILES>>	Все файлы из файловой системы

Например, приведенная ниже запись полномочий означает, что доступ разрешается ко всем файлам в каталоге /myapp и любом из его подкаталогов.

```
permission java.io.FilePermission
           "/myapp/-", "read,write,delete";
```

Для обозначения обратной косой черты в пути к файлам в Windows этот знак следует экранировать, указав его дважды:

```
permission java.io.FilePermission
           "c:\\myapp\\-", "read,write,delete";
```

Целевые объекты прав доступа через сетевой сокет требуют указания хоста (т.е. сетевого узла) и диапазона портов. Синтаксическая форма для указания хоста выглядит следующим образом:

имя_хоста или ***IP-адрес***	Одиночный хост
localhost или пустая строка	Локальный хост
.суффикс_домена	Любой хост, принадлежащий домену, имя которого оканчивается указанным суффиксом
*	Все хосты

Номера портов являются необязательными и указываются в приведенной ниже форме.

:n	Единственный порт
:n-	Все порты с номерами ***n*** и выше
:-n	Все порты с номерами ***n*** и ниже
:n1-n2	Все порты в пределах от ***n1*** до ***n2***

Ниже приведен пример записи прав доступа через сетевой сокет.

```
permission java.net.SocketPermission
           "*.horstmann.com:8000-8999", "connect";
```

Наконец, целевые объекты прав доступа к свойствам могут принимать одну из двух следующих форм:

свойство	Отдельное свойство
префикс_свойства.*	Все свойства с указанным суффиксом

Таким образом, права доступа к свойствам могут выглядеть как "java.home" и как "java.vm.*". Например, следующая запись полномочий означает, что программе разрешается считывать все свойства, начинающиеся с java.vm:

```
permission java.util.PropertyPermission "java.vm.*", "read";
```

В файлах правил защиты допускается использовать и системные свойства. Лексема ${*свойство*} в этом случае заменяется значением свойства. Например, лексема ${user.home} заменяется основным каталогом пользователя. Ниже приведен типичный пример применения системного свойства user.home в записи полномочий.

```
permission java.io.FilePermission
           "${user.home}", "read,write";
```

Чтобы упростить дело при составлении не зависящих от используемой платформы файлов правил защиты, вместо явных разделителей / или \\ рекомендуется использовать свойство file.separator или даже его сокращенный вариант ${/}. Например, для предоставления прав на чтение и запись в основном каталоге пользователя и всех его подкаталогах можно воспользоваться следующей записью, удобной с точки зрения переносимости:

```
permission java.io.FilePermission
           "${user.home}${/}-", "read,write";
```

10.2.4. Специальные полномочия

В этом разделе будет показано, каким образом создаются собственные классы полномочий, на которые пользователи могут ссылаться в файлах правил защиты. Для реализации классов полномочий следует расширить класс `Permission` и реализовать перечисленные ниже методы.

- Конструктор с двумя строковыми параметрами для указания объекта и списка действий.
- Метод `String getActions()`.
- Метод `boolean equals(Object other)`.
- Метод `int hashCode()`.
- Метод `boolean implies(Permission other)`.

Последний метод самый важный. Для полномочий предусмотрен определенный *порядок*, в соответствии с которым наиболее общие полномочия *подразумевают* использование специальных прав доступа. Например, приведенное ниже право доступа к файлу разрешает чтение и запись любого файла из каталога `/tmp` и любых его подкаталогов.

```
p1 = new FilePermission("/tmp/-", "read, write");
```

Это общее право доступа подразумевает наличие других, специальных прав доступа:

```
p2 = new FilePermission("/tmp/-", "read");
p3 = new FilePermission("/tmp/aFile", "read, write");
p4 = new FilePermission("/tmp/aDirectory/-", "write");
```

Иными словами, право доступа к файлу `p1` подразумевает наличие другого права доступа `p2`, если выполняются следующие условия.

3. Набор целевых файлов в `p1` содержит набор целевых файлов в `p2`.

4. Набор действий в `p1` содержит набор действий в `p2`.

Рассмотрим в качестве примера применение метода `implies()`. Если конструктор класса `FileInputStream()` открывает файл для чтения, то он проверяет наличие прав такого доступа. Для выполнения этой проверки *специальный* объект прав доступа передается методу `checkPermission()`, как показано ниже.

```
checkPermission(new FilePermission(fileName, "read"));
```

После этого диспетчер защиты запрашивает все имеющиеся полномочия, подразумевается ли в них специальное право доступа. И если это право доступа подразумевается в любом из них, то оно проходит проверку. В частности, полномочия типа `AllPermission` подразумевают все права доступа.

Определяя собственные классы полномочий, необходимо также обозначить соответственно, что именно подразумевается для объектов полномочий. Например, для определения прав доступа пользователя `Tommy` к телевизору под управлением технологии Java в некотором промежутке времени можно определить следующий объект класса `TVPermission`:

```
new TVPermission("Tommy:2-12:1900-2200", "watch,record")
```

Этот класс разрешает пользователю Tommy смотреть и записывать телевизионные передачи на каналах 2–12 в период времени от 19:00 до 22:00. Для применения приведенного ниже специального права доступа придется реализовать метод implies().

```
new TVPermission("Tommy:4:2000-2100", "watch")
```

10.2.5. Реализация класса полномочий

В этом разделе на примере конкретной программы демонстрируется реализация новых полномочий для контроля над текстом, вставляемым в текстовую область. В обязанности этой программы входит предотвращение попыток ввода в текстовую область всевозможных "непристойных слов" вроде sex, drugs и C++, но не rock-n-roll! Чтобы предоставить возможность передавать список всех подобных непристойных слов в файл правил защиты, применяется специальный класс полномочий. Он является производным от класса JTextArea и всегда запрашивает диспетчер защиты, можно ли вводить новый текст в текстовую область, как показано ниже. Если диспетчер защиты предоставляет полномочия типа WordCheckPermission, новый текст вводится, а иначе метод checkPermission() генерирует исключение.

```
class WordCheckTextArea extends JTextArea
{
  public void append(String text)
  {
    var p = new WordCheckPermission(text, "insert");
    SecurityManager manager = System.getSecurityManager();
    if (manager != null) manager.checkPermission(p);
    super.append(text);
  }
}
```

Полномочия на проверку слов (WordCheckPermission) допускают выполнение двух следующих действий: insert (вставка указанного текста) и avoid (ввод текста без указанных непристойных слов). Запускать рассматриваемую здесь программу следует с помощью приведенного ниже файла правил защиты. В этом файле предоставляется разрешение на вставку любого текста, который не содержит такие непристойные слова, как sex, drugs и C++.

```
grant
{
  permission permissions.WordCheckPermission
             "sex,drugs,C++", "avoid";
};
```

При разработке класса WordCheckPermission особое внимание следует уделить методу implies(). Ниже перечислены правила, определяющие, должны ли полномочия p1 предполагать полномочия p2.

- Если полномочия p1 допускают действие avoid, а полномочия p2 — действие insert, то целевой объект с полномочиями p2 должен исключать все слова из полномочий p1. Например, следующие полномочия:

  ```
  permissions.WordCheckPermission "sex, drugs, C++", "avoid"
  ```

- предполагают такие полномочия:

```
permissions.WordCheckPermission
            "Mary had a little lamb", "insert"
```

- Если оба вида полномочий, p1 и p2, допускают действие avoid, то набор слов с полномочиями p2 должен содержать все слова из набора слов с полномочиями p1. Например, следующие полномочия:

```
permissions.WordCheckPermission "sex,drugs", "avoid"
```

- предполагают такие полномочия:

```
permissions.WordCheckPermission "sex,drugs,C++", "avoid"
```

- Если оба вида полномочий p1 и p2 допускают действие insert, то текст с полномочиями p1 должен содержать текст с полномочиями p2. Например, следующие полномочия:

```
permissions.WordCheckPermission
            "Mary had a little lamb", "insert"
```

- предполагают такие полномочия:

```
permissions.WordCheckPermission "a little lamb", "insert"
```

Исходный код, реализующий класс WordCheckPermission, представлен в листинге 10.4. Следует заметить, что целевой объект полномочий извлекается методом с не совсем подходящим именем getName() из класса Permission.

Полномочия описываются в файлах правил защиты с помощью пары символьных строк, поэтому классы прав доступа должны быть подготовлены к синтаксическому анализу этих строк. Так, в рассматриваемом здесь примере для преобразования списка разделяемых запятыми непристойных слов с полномочиями avoid в подлинное множество типа Set используется следующий метод:

```
public Set<String> badWordSet()
{
  Set<String> set = new HashSet<String>();
  set.addAll(Arrays.asList(getName().split(",")));
  return set;
}
```

Этот метод позволяет использовать для сравнения множеств (в данном случае — наборов слов) методы equals() и containsAll(). Как было показано в главе 9 первого тома настоящего издания, метод equals() из класса Set признает два множества равными в том случае, если в них содержатся одинаковые элементы, независимо от порядка их расположения. Например, наборы слов "sex, drugs, C++" и "C++, drugs, sex" будут признаны равными.

ВНИМАНИЕ! Класс полномочий должен быть открытым (**public**). Загрузчик файлов правил защиты не может загружать классы с уровнем доступности на уровне пакета, поэтому он проигнорирует любые классы, которые ему не удастся обнаружить.

В исходном коде, приведенном в листинге 10.5, показано, каким образом действует класс WordCheckPermission. Попробуйте ввести в текстовой области какой-нибудь текст и щелкнуть на кнопке Insert (Вставить). Если проверка

на безопасность пройдет успешно, текст будет вставлен в текстовую область, в противном случае появится сообщение об ошибке (рис. 10.7).

Рис. 10.7. Программа **PermissionTest** в действии

На этом рассмотрение способов и средств настройки безопасности на платформе Java завершается. Как правило, вам придется лишь подкорректировать должным образом стандартные полномочия. Но если потребуются дополнительные средства для контроля безопасности, то вы сможете теперь определить специальные классы полномочий и настроить их таким же образом, как и стандартные полномочия.

Листинг 10.4. Исходный код из файла **permissions/WordCheckPermission.java**

```
1  package permissions;
2
3  import java.security.*;
4  import java.util.*;
5
6  /**
7   * Полномочия на проверку непристойных слов
8  */
9  public class WordCheckPermission extends Permission
10 {
11   private String action;
12
13   /**
14    * Конструирует полномочия на проверку
15    * непристойных слов
16    * @param target Список разделяемых запятыми слов
17    * @param anAction Действие "вставить" или "исключить"
18    */
19   public WordCheckPermission(
20         String target, String anAction)
21   {
```

```
22      super(target);
23      action = anAction;
24    }
25    public String getActions()
27    {
28      return action;
29    }
30
31    public boolean equals(Object other)
32    {
33      if (other == null) return false;
34      if (!getClass().equals(other.getClass()))
35        return false;
36      var b = (WordCheckPermission) other;
37      if (!Objects.equals(action, b.action))
38        return false;
39      if ("insert".equals(action))
40        return Objects.equals(getName(), b.getName());
42      else if ("avoid".equals(action)) return
43         badWordSet().equals(b.badWordSet());
44      else return false;
45    }
46
47    public int hashCode()
48    {
49      return Objects.hash(getName(), action);
50    }
51
52    public boolean implies(Permission other)
53    {
54      if (!(other instanceof WordCheckPermission))
55        return false;
56      var b = (WordCheckPermission) other;
57      if (action.equals("insert"))
58      {
59        return b.action.equals("insert")
60            && getName().indexOf(b.getName()) >= 0;
61      }
62      else if (action.equals("avoid"))
63      {
64        if (b.action.equals("avoid")) return
65          b.badWordSet().containsAll(badWordSet());
66        else if (b.action.equals("insert"))
67        {
68          for (String badWord : badWordSet())
69            if (b.getName().indexOf(badWord) >= 0)
70              return false;
71          return true;
72        }
73        else return false;
74      }
75      else return false;
76    }
77
78    /**
79     * Получает непристойные слова, описываемые
```

```
80     * данным правилом прав доступа
81     * @return Непристойные слова
82    */
83    public Set<String> badWordSet()
84    {
85      Set<String> set = new HashSet<>();
86      set.addAll(Arrays.asList(getName().split(",")));
87      return set;
88    }
89 }
```

Листинг 10.5. Исходный код из файла **permissions/PermissionTest.java**

```
1  package permissions;
2
3  import java.awt.*;
4
5  import javax.swing.*;
6
7  /**
8   * Этот класс демонстрирует применение специальных
9   * полномочий типа WordCheckPermission
10  * @version 1.05 2018-05-01
11  * @author Cay Horstmann
12  */
13 public class PermissionTest
14 {
15   public static void main(String[] args)
16   {
17     System.setProperty("java.security.policy",
18           "permissions/PermissionTest.policy");
19     System.setSecurityManager(new SecurityManager());
20     EventQueue.invokeLater(() ->
21       {
22         var frame = new PermissionTestFrame();
23         frame.setTitle("PermissionTest");
24         frame.setDefaultCloseOperation(
25               JFrame.EXIT_ON_CLOSE);
26         frame.setVisible(true);
27       });
28   }
29 }
30
31 /**
32  * Этот фрейм содержит текстовое поле для ввода слов
33  * в текстовой области, защищенной от вставки
34  * непристойных слов
35  */
36 class PermissionTestFrame extends JFrame
37 {
38   private JTextField textField;
39   private WordCheckTextArea textArea;
40   private static final int TEXT_ROWS = 20;
41   private static final int TEXT_COLUMNS = 60;
42
```

```
43   public PermissionTestFrame()
44   {
45     textField = new JTextField(20);
46     var panel = new JPanel();
47     panel.add(textField);
48     var openButton = new JButton("Insert");
49     panel.add(openButton);
50     openButton.addActionListener(event ->
51                insertWords(textField.getText()));
52
53     add(panel, BorderLayout.NORTH);
54
55     textArea = new WordCheckTextArea();
56     textArea.setRows(TEXT_ROWS);
57     textArea.setColumns(TEXT_COLUMNS);
58     add(new JScrollPane(textArea), BorderLayout.CENTER);
59     pack();
60   }
61
62   /**
63    * Пытается вставить слова в текстовую область.
64    * Отображает диалоговое окно, если попытка
65    * окажется неудачной
66    * @param words Вставляемые слова
67    */
68   public void insertWords(String words)
69   {
70     try
71     {
72       textArea.append(words + "\n");
73     }
74     catch (SecurityException ex)
75     {
76       JOptionPane.showMessageDialog(this,
77                   "I am sorry, but I cannot do that.");
78       ex.printStackTrace();
79     }
80   }
81 }
82
83 /**
84  * Текстовая область, в которой метод ввода текста
85  * проверяет в целях безопасности, чтобы в нее не
86  * были вставлены непристойные слова
87  */
88 class WordCheckTextArea extends JTextArea
89 {
90   public void append(String text)
91   {
92     var p = new WordCheckPermission(text, "insert");
93     SecurityManager manager = System.getSecurityManager();
94     if (manager != null) manager.checkPermission(p);
95     super.append(text);
96   }
97 }
```

java.security.Permission 1.2

- **Permission(String *name*)**
  Создает полномочия с указанным именем целевого объекта.
- **String getName()**
  Возвращает имя целевого объекта для данных полномочий.
- **boolean implies(Permission *other*)**
  Проверяет, предполагают ли данные полномочия другие полномочия. Это имеет место в том случае, если в других полномочиях описывается более конкретное условие, вытекающее из условия, указанного в данных полномочиях.

10.3. Аутентификация пользователей

В прикладном интерфейсе Java API предоставляется каркас под названием JAAS (Java Authentication and Authorization Service — служба аутентификации и авторизации в Java). Он позволяет сочетать аутентификацию, предоставляемую на отдельной платформе, с управлением правами доступа и подробно рассматривается в последующих разделах.

10.3.1. Каркас JAAS

Как следует из названия каркаса JAAS, он состоит из двух компонентов. В частности, компонент аутентификации отвечает за опознавание пользователей программ, а компонент авторизации — за проверку их полномочий.

Каркас JAAS, по существу, представляет собой встраиваемый прикладной интерфейс API, отделяющий прикладные программы на Java от конкретной технологии, применяемой для реализации средств аутентификации. Помимо прочего, в нем поддерживаются механизмы регистрации в UNIX и Windows, а также механизмы аутентификации Kerberos и по сертификатам.

После аутентификации за пользователем может быть закреплен определенный набор полномочий. Ниже приведен пример, в котором пользователю `harry` предоставляется особый ряд полномочий, которых нет ни у кого из других пользователей.

```
grant principal com.sun.security.auth.UnixPrincipal "harry"
{
  permission java.util.PropertyPermission "user.*", "read";
  . . .
};
```

В данном примере класс `com.sun.security.auth.UnixPrincipal` выполняет проверку имени пользователя UNIX, запускающего программу. Метод `getName()` из этого класса возвращает имя пользователя, которое зарегистрировано в системе UNIX и сравнивается на равенство с именем `harry`.

Класс `LoginContext` дает диспетчеру защиты возможность проверить правильность предоставления таких полномочий. Ниже приведена общая структура кода регистрации.

```
try
{
  System.setSecurityManager(new SecurityManager());
  // определяется в конфигурационном файле JAAS:
  var context = new LoginContext("Login1");
  context.login();
  // получить аутентифицированный объект типа Subject:
  Subject subject = context.getSubject();
  . . .
  context.logout();
}
catch (LoginException exception) // это исключение
       // генерируется при неудачной попытке регистрации
{
  exception.printStackTrace();
}
```

Теперь объект `subject` представляет прошедшего аутентификацию пользователя. Строковый параметр `"Login1"` в конструкторе класса `LoginContext` обозначает запись с аналогичным именем в конфигурационном файле JAAS. Ниже приведен пример такого конфигурационного файла.

```
Login1
{
  com.sun.security.auth.module.UnixLoginModule required;
  com.whizzbang.auth.module.RetinaScanModule sufficient;
};

Login2
{
  . . .
};
```

Разумеется, в JDK не предусмотрено никаких модулей для биометрической регистрации. В состав пакета `com.sun.security.auth.module` входят только следующие модули:

```
UnixLoginModule
NTLoginModule
Krb5LoginModule
JndiLoginModule
KeyStoreLoginModule
```

Правила регистрации состоят из ряда модулей, каждый из которых обозначен как `required`, `sufficient`, `requisite` или `optional`. Значение этих ключевых слов можно понять из приведенного ниже описания алгоритма регистрации.

При регистрации выполняется аутентификация *субъекта*, который может иметь несколько *принципалов*. Каждый принципал описывает какое-то свойство субъекта, например, имя пользователя, идентификатор группы или роль. Как было показано ранее в операторе `grant`, принципалы управляют правами доступа. Объект типа `com.sun.security.auth.UnixPrincipal` описывает имя пользователя, зарегистрированное в UNIX, а объект типа `UnixNumeric Group Principal` может выполнять проверку на принадлежность пользователя к группе в UNIX. Предложение `grant` дает возможность проверить наличие принципала с помощью следующей синтаксической конструкции:

```
grant КлассПринципала "ИмяПринципала"
```

Следовательно, в рассматриваемом здесь примере это предложение принимает следующий вид:

```
grant com.sun.security.auth.UnixPrincipal "harry"
```

Когда пользователь регистрируется, в отдельном контексте управления доступом должен быть выполнен код, требующий проверки принципалов. Для инициирования нового привилегированного действия типа `PriviligedAction` вызывается статический метод `doAs()` или `doAsPrivileged()`.

Оба эти метода выполняют действие, вызывая метод `run()` для объекта, класс которого реализует интерфейс `PrivilegedAction`, а также используя полномочия принципалов субъекта, как показано ниже. Если же при выполнении действий генерируются проверяемые исключения, то вместо упомянутого выше интерфейса лучше реализовать интерфейс `PriviligedExceptionAction`.

```
PrivilegedAction<T> action = () ->
{
  // выполнить код с полномочиями принципалов субъекта
  . . .
};
T result = Subject.doAs(subject, action);
    // или Subject.doAsPrivileged(subject, action, null)
```

Методы `doAs()` и `doAsPrivileged()` отличаются лишь незначительно. Так, метод `doAs()` запускается в текущем контексте управления доступом, а метод `doAsPrivileged()` — в новом контексте. Кроме того, метод `doAsPrivileged()` позволяет разделять права доступа для кода регистрации и бизнес-логики. В рассматриваемом здесь примере прикладной программы код регистрации имеет следующие полномочия:

```
permission javax.security.auth.AuthPermission
                          "createLoginContext.Login1";
permission javax.security.auth.AuthPermission
                          "doAsPrivileged";
```

Пользователь, прошедший аутентификацию, получает приведенные ниже полномочия. Если бы вместо метода `doAsPrivileged()` использовался метод `doAs()`, такие же полномочия требовались бы и коду регистрации!

```
permission java.util.PropertyPermission "user.*", "read";
```

В рассматриваемом здесь примере программы `AuthTest` (ее исходный код приведен в листингах 10.6 и 10.7) демонстрируется, как ограничивать права доступа определенных пользователей. Эта программа аутентифицирует пользователя, а затем выполняет простое действие, извлекающее системное свойство. Чтобы данный пример программы оказался работоспособным, прикладной код для регистрации и выполнения указанного действия следует упаковать в два отдельных архивных JAR-файла, выполнив следующие команды:

```
javac auth/*.java
jar cvf login.jar auth/AuthTest.class
jar cvf action.jar auth/SysPropAction.class
```

Как следует из файла правил защиты, содержимое которого приведено в листинге 10.8, пользователь системы UNIX с именем `harry` обладает правами

на чтение всех файлов. Замените сначала имя `harry` своим учетным именем, а затем выполните приведенную ниже команду. Настройка регистрации представлена в листинге 10.9.

```
java -classpath login.jar:action.jar \
     -Djava.security.policy=auth/AuthTest.policy \
     -Djava.security.auth.login.config=auth/jaas.config \
     auth.AuthTest
```

В системе Windows следует заменить `UnixPrincipal` на `NTUserPrincipal` в файлах `AuthTest.policy` и `jaas.configUnix`, а также использовать точку с запятой для разделения архивных JAR-файлов, как показано в приведенной ниже команде.

```
java -classpath login.jar;action.jar . . .
```

После этого программа `AuthTest` должна отобразить значение свойства `user.home`. Но если зарегистрироваться под другим учетным именем, то должно возникнуть исключение в связи с отсутствием требующихся полномочий.

ВНИМАНИЕ! Очень важно выполнить *все* приведенные выше инструкции точно и аккуратно, поскольку даже незначительные отклонения могут привести к неверной настройке аутентификации пользователей.

Листинг 10.6. Исходный код из файла **`auth/AuthTest.java`**

```
1  package auth;
2
3  import javax.security.auth.*;
4  import javax.security.auth.login.*;
5
6  /**
7   * В этой программе демонстрируется аутентификация
9   * пользователей через специальную регистрацию и
10  * последующее выполнение действия типа SysPropAction
11  * привилегиями зарегистрированного пользователя
12  * @version 1.02 2018-05-01
13  * @author Cay Horstmann
14  */
15 public class AuthTest
16 {
17   public static void main(final String[] args)
18   {
19     System.setSecurityManager(new SecurityManager());
20     try
21     {
22       var context = new LoginContext("Login1");
23       context.login();
24       System.out.println("Authentication successful.");
25       Subject subject = context.getSubject();
26       System.out.println("subject=" + subject);
27       var action = new SysPropAction("user.home");
28       String result = Subject.doAsPrivileged(
29                       subject, action, null);
```

```
30        System.out.println(result);
31        context.logout();
32      }
33      catch (LoginException e)
34      {
35        e.printStackTrace();
36      }
37    }
38 }
```

Листинг 10.7. Исходный код из файла **auth/SysPropAction.java**

```
1  package auth;
2
3  import java.security.*;
4
5  /**
6   * Это действие осуществляет поиск системного свойства
7   * @version 1.01 2007-10-06
8   * @author Cay Horstmann
9  */
10 public class SysPropAction
11        implements PrivilegedAction<String>
12 {
13   private String propertyName;
14
15   /**
16    * Конструирует действие для поиска заданного свойства
17    * @param propertyName Имя свойства (например, "user.home")
18    */
19   public SysPropAction(String propertyName)
20   {
21     this.propertyName = propertyName;
22   }
23
24   public String run()
25   {
26     return System.getProperty(propertyName);
27   }
28 }
```

Листинг 10.8. Исходный код из файла **auth/AuthTest.policy**

```
1  grant codebase "file:login.jar"
2  {
3    permission javax.security.auth.AuthPermission
4                              "createLoginContext.Login1";
5    permission javax.security.auth.AuthPermission
6                              "doAsPrivileged";
7  };
8
9  grant principal com.sun.security.auth.UnixPrincipal
10                                  "harry"
```

```
11 {
12   permission java.util.PropertyPermission "user.*", "read";
13 };
```

Листинг 10.9. Исходный код из файла **auth/jaas.config**

```
1 Login1
2 {
3   com.sun.security.auth.module.UnixLoginModule required;
4 };
```

javax.security.auth.login.LoginContext 1.4

- **LoginContext(String *name*)**

  Создает контекст регистрации. Параметр ***name*** соответствует дескриптору регистрации в конфигурационном файле службы JAAS.
- **void login()**

  Регистрирует субъект, а при неудачном исходе регистрации генерирует исключение типа **LoginException**. Вызывает метод **login()** для диспетчеров, указанных в конфигурационном файле JAAS.
- **void logout()**

  Отменяет регистрацию субъекта. Вызывает метод **logout()** для диспетчеров, указанных в конфигурационном файле JAAS.
- **Subject getSubject()**

  Возвращает аутентифицированный субъект.

javax.security.auth.Subject 1.4

- **Set<Principal> getPrincipals()**

  Возвращает принципалы данного субъекта.
- **static Object doAs(Subject *subject*, PrivilegedAction *action*)**
- **static Object doAs(Subject *subject*, PrivilegedExceptionAction *action*)**
- **static Object doAsPrivileged(Subject *subject*, PrivilegedAction *action*, AccessControlContext *context*)**
- **static Object doAsPrivileged(Subject *subject*, PrivilegedExceptionAction *action*, AccessControlContext *context*)**

  Выполняют привилегированное действие от имени субъекта. Возвращают объект, сформированный методом **run()** из класса, реализующего интерфейс **PrivilegedAction**. Методы **doAsPrivileged()** выполняют действие в указанном контексте управления доступом. Для них можно указать "моментальный снимок" контекста, полученный ранее в результате вызова метода **AccessController.getContext()**. Если задано пустое значение **null**, код будет выполняться в новом контексте.

java.security.PrivilegedAction 1.4

- **Object run()**
  Этот метод необходимо определить самостоятельно для выполнения кода от имени субъекта.

java.security.PrivilegedExceptionAction 1.4

- **Object run()**
  Этот метод необходимо определить самостоятельно для выполнения кода от имени субъекта. Он может генерировать любые проверяемые исключения.

java.security.Principal 1.1

- **String getName()**
  Возвращает имя данного принципала.

10.3.2. Модули регистрации JAAS

В этом разделе рассматривается пример применения JAAS, демонстрирующий следующее:

- как реализовать свой собственный модуль регистрации;
- как реализовать *ролевую* аутентификацию.

Предоставление собственного модуля регистрации бывает полезным в тех случаях, когда учетные данные сохраняются в базе данных. Даже если вас вполне устраивает стандартный модуль регистрации, изучение процесса реализации специального модуля поможет вам лучше разобраться с предназначением параметров в конфигурационном файле JAAS.

Реализация механизма ролевой аутентификации играет важную роль в тех случаях, когда требуется управлять большим количеством пользователей. Размещать имена всех допустимых пользователей в файле правил регистрации непрактично. Вместо этого лучше поступить таким образом, чтобы модуль регистрации сопоставлял пользователей с ролями вроде "admin" или "HR" и предоставлял им права, исходя из их ролей.

Одна из задач модуля регистрации состоит в заполнении множества принципалов аутентифицируемого субъекта. Если в модуле регистрации поддерживаются роли, он должен также вводить в это множество объекты типа `Principal`, описывающие соответствующие роли. К сожалению, в библиотеке Java отсутствует класс для решения этой задачи, поэтому нам пришлось создать собственный класс, исходный код которого приведен в листинге 10.10. Этот класс просто сохраняет пары "описание–значение" типа `role=admin`. А его метод `getName()` возвращает их, что дает возможность вводить ролевые полномочия в файл правил регистрации, как показано ниже.

```
grant principal SimplePrincipal "role=admin" { . . . }
```

Рассматриваемый здесь модуль регистрации предусматривает поиск имен, паролей и ролей пользователей в текстовом файле, содержащем строки, аналогичные приведенным ниже. Разумеется, при разработке реального модуля регистрации эти сведения следовало бы хранить в какой-нибудь базе данных или словаре.

```
harry|secret|admin
carl|guessme|HR
```

Исходный код рассматриваемого здесь примера модуля регистрации типа `SimpleLoginModule` представлен в листинге 10.11. Метод `checkLogin()` проверяет, соответствуют ли имя пользователя и пароль какой-нибудь записи в файле паролей. Если они соответствуют, то во множество принципалов данного субъекта вводятся два объекта типа `SimplePrincipal`:

```
Set<Principal> principals = subject.getPrincipals();
principals.add(new SimplePrincipal("username", username));
principals.add(new SimplePrincipal("role", role));
```

Остальная часть модуля типа `SimpleLoginModule` довольно проста. В частности, метод `initialize()` получает в качестве параметров следующее.

- Аутентифицируемый объект типа `Subject`.
- Обработчик для извлечения учетных данных.
- Отображение `sharedState`, которое можно использовать для обмена данными между модулями регистрации.
- Отображение `options`, которое содержит пары "имя–значение", задаваемые при настройке модуля регистрации.

Допустим, рассматриваемый здесь модуль регистрации настраивается приведенным ниже образом. В таком случае этот модуль будет извлекать из отображения `options` параметры настройки `pwfile`.

```
SimpleLoginModule required pwfile="password.txt";
```

Сбором имени пользователя и пароля данный модуль регистрации не занимается, поскольку эта задача поручена отдельному обработчику. Такое разделение ответственности позволяет использовать один и тот же модуль, не особенно беспокоясь, откуда именно поступают учетные данные: из диалогового окна ГПИ, командной строки консоли или из конфигурационного файла. Требующийся обработчик задается при создании экземпляра класса `LoginContext`, как показано в приведенном ниже примере кода.

```
var = new LoginContext("Login1", new com.sun.security
          .auth.callback.DialogCallbackHandler());
```

Обработчик типа `DialogCallbackHandler` открывает простое диалоговое окно в графическом пользовательском интерфейсе для запрашивания имени пользователя и пароля, а обработчик типа `com.sun.security.auth.callback.TextCallbackHandler` получает эти учетные данные с консоли.

Но в данном примере для получения имени пользователя и пароля используется собственный графический пользовательский интерфейс (рис. 10.8). Для этой цели создается специальный обработчик, способный сохранять и возвращать учетные данные, как демонстрируется в листинге 10.12.

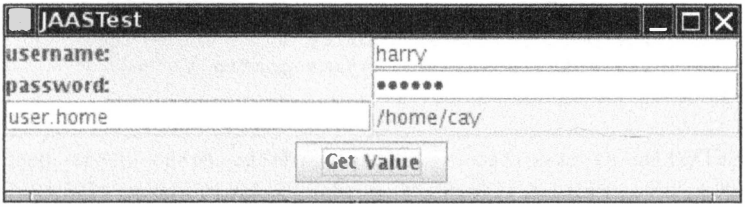


Рис. 10.8. Специальный модуль регистрации

Этот обработчик состоит из единственного метода `handle()`, который обрабатывает массив объектов типа `Callback`. Интерфейс `Callback` реализуется рядом предопределенных классов вроде `NameCallback` и `PasswordCallback`. По желанию можно добавить к ним собственный класс, например `RetinaScanCallback`. Исходный код этого обработчика выглядит не очень изящно, поскольку он нуждается в анализе типа объектов обратного вызова, как показано ниже.

```
public void handle(Callback[] callbacks)
{
  for (Callback callback : callbacks)
  {
    if (callback instanceof NameCallback) . . .
    else if (callback instanceof PasswordCallback) . . .
    else . . .
  }
}
```

Модуль регистрации сначала подготавливает массив объектов типа `Callback`, которые требуются ему для аутентификации, как следует из приведенного ниже фрагмента кода, а затем извлекает из этих объектов всю необходимую информацию.

```
var nameCall = new NameCallback("username: ");
var passCall = new PasswordCallback("password: ", false);
callbackHandler.handle(new Callback[]
{ nameCall, passCall });
```

При выполнении примера программы из листинга 10.13 отображается форма для ввода учетных данных и имени системного свойства. При удачном исходе аутентификации пользователя значение этого свойства извлекается в объект типа `PrivilegedAction`. Как следует из файла правил регистрации, содержимое которого приведено в листинге 10.14, правами на чтение свойств обладают только пользователи с ролью `admin`.

Как и в примере программы из предыдущего раздела, код регистрации и код действия требуется разделить. Для этого создаются два архивных JAR-файла:

```
javac *.java
jar cvf login.jar JAAS*.class Simple*.class
jar cvf action.jar SysPropAction.class
```

После этого рассматриваемая здесь программа запускается на выполнение по приведенной ниже команде. Содержимое конфигурационного файла представлено в листинге 10.15.

```
java -classpath login.jar:action.jar \
     -Djava.security.policy=JAASTest.policy \
     -Djava.security.auth.login.config=jaas.config \
     JAASTest
```

НА ЗАМЕТКУ! Можно также обеспечить поддержку более сложного двухэтапного протокола и с его помощью сделать так, чтобы регистрация считалась *завершенной* только при успешном выполнении всех модулей, предусмотренных в настройке регистрации. Подробнее об этом можно узнать из руководства для разработчиков модулей регистрации, доступного по адресу **https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASLMDevGuide.html**.

Листинг 10.10. Исходный код из файла **jaas/SimplePrincipal.java**

```
1  package jaas;
2
3  import java.security.*;
4  import java.util.*;
5
6  /**
7   * Принципал с именованным значением (например,
8   * "role=HR" или "username=harry")
9  */
10 public class SimplePrincipal implements Principal
11 {
12   private String descr;
13   private String value;
14
15   /**
16    * Конструирует принципал типа SimplePrincipal
17    * для хранения отдельного описания и значения
18    * @param descr Описание
19    * @param value Связанное с ним значение
20   */
21   public SimplePrincipal(String descr, String value)
22   {
23     this.descr = descr;
24     this.value = value;
25   }
26
27   /**
28    * Возвращает имя роли данного принципала
29    * @return Имя роли
30   */
31   public String getName()
32   {
33     return descr + "=" + value;
34   }
35
36   public boolean equals(Object otherObject)
37   {
38     if (this == otherObject) return true;
39     if (otherObject == null) return false;
40     if (getClass() != otherObject.getClass())
```

```
41        return false;
43      var other = (SimplePrincipal) otherObject;
44      return Objects.equals(getName(), other.getName());
45    }
46
47    public int hashCode()
48    {
49      return Objects.hashCode(getName());
50    }
51  }
```

Листинг 10.11. Исходный код из файла **jaas/SimpleLoginModule.java**

```
1   package jaas;
2
3   import java.io.*;
4   import java.nio.file.*;
5   import java.security.*;
6   import java.util.*;
7   import javax.security.auth.*;
8   import javax.security.auth.callback.*;
9   import javax.security.auth.login.*;
10  import javax.security.auth.spi.*;
11
12  /**
13   * Этот модуль регистрации аутентифицирует
14   * пользователей, считывая их имена, пароли
15   * и роли из текстового файла
16   */
17  public class SimpleLoginModule implements LoginModule
18  {
19    private Subject subject;
20    private CallbackHandler callbackHandler;
21    private Map<String, ?> options;
22
23    public void initialize(Subject subject,
24                CallbackHandler callbackHandler,
25                Map<String, ?> sharedState,
26                Map<String, ?> options)
27    {
28      this.subject = subject;
29      this.callbackHandler = callbackHandler;
30      this.options = options;
31    }
32
33    public boolean login() throws LoginException
34    {
35      if (callbackHandler == null)
36        throw new LoginException("no handler");
37
38      var nameCall = new NameCallback("username: ");
39      var passCall = new PasswordCallback(
40                          "password: ", false);
41      try
42      {
```

```
43        callbackHandler.handle(new Callback[]
44        { nameCall, passCall });
45      }
46      catch (UnsupportedCallbackException e)
47      {
48        var e2 = new LoginException(
49                    "Unsupported callback");
50        e2.initCause(e);
51        throw e2;
52      }
53      catch (IOException e)
54      {
55        LoginException e2 = new LoginException(
56                              "I/O exception in callback");
57        e2.initCause(e);
58        throw e2;
59      }
60
61      try
62      {
63        return checkLogin(nameCall.getName(),
64                          passCall.getPassword());
65      }
66      catch (IOException ex)
67      {
68        LoginException ex2 = new LoginException();
69        ex2.initCause(ex);
70        throw ex2;
71      }
72    }
73
74    /**
75     * Проверяет достоверность данных аутентификации.
76     * Если они достоверны, то субъекту требуются
77     * принципалы для имени пользователя и его роли
78     * @param username Имя пользователя
79     * @param password Символьный массив, содержащий пароль
80     * @return Логическое значение true, если
81     *         данные достоверны
83     */
84    private boolean checkLogin(
85            String username, char[] password)
86            throws LoginException, IOException
87    {
88      try (Scanner in = new Scanner(Paths.get(
89           "" + options.get("pwfile")), "UTF-8"))
90    {
91        while (in.hasNextLine())
92        {
93          String[] inputs = in.nextLine().split("\\|");
94          if (inputs[0].equals(username)
95              && Arrays.equals(inputs[1].toCharArray(),
96                               password))
97          {
98            String role = inputs[2];
```

```
99            Set<Principal> principals =
100                           subject.getPrincipals();
101           principals.add(new SimplePrincipal(
102                           "username", username));
103           principals.add(new SimplePrincipal(
104                           "role", role));
105           return true;
106         }
107       }
108       return false;
109     }
110   }
111
112   public boolean logout()
113   {
114     return true;
115   }
116
117   public boolean abort()
118   {
119     return true;
120   }
121
122   public boolean commit()
123   {
124     return true;
125   }
126 }
```

10.12. Исходный код из файла **aas/SimpleCallbackHandler.java**

```
1  package jaas;
2
3  import javax.security.auth.callback.*;
4
5  /**
6   * Этот простой обработчик обратных вызовов
7   * предоставляет заданное имя пользователя и пароль
8   */
9  public class SimpleCallbackHandler
10        implements CallbackHandler
11 {
12   private String username;
13   private char[] password;
14
15   /**
16    * Конструирует обработчик обратных вызовов
17    * @param username Имя пользователя
18    * @param password Символьный массив, содержащий пароль
19   */
20   public SimpleCallbackHandler(String username, char[] password)
21   {
22     this.username = username;
23     this.password = password;
```

```
24    }
25
26    public void handle(Callback[] callbacks)
27    {
28       for (Callback callback : callbacks)
29       {
30          if (callback instanceof NameCallback)
31          {
32             ((NameCallback) callback).setName(username);
33          }
34          else if (callback instanceof PasswordCallback)
35          {
36             ((PasswordCallback) callback).setPassword(password);
37          }
38       }
39    }
40 }
```

Листинг 10.13. Исходный код из файла **jaas/JAASTest.java**

```
1  package jaas;
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  /**
7   * В этой программе сначала производится аутентификация
8   * пользователя через специальную регистрацию, а затем
9   * поиск системного свойства с назначенными для
10  * пользователя привилегиями
11  * @version 1.03 2018-05-01
12  * @author Cay Horstmann
13  */
14 public class JAASTest
15 {
16    public static void main(final String[] args)
17    {
18       System.setSecurityManager(new SecurityManager());
19       EventQueue.invokeLater(() ->
20          {
21             var frame = new JAASFrame();
22             frame.setDefaultCloseOperation(
23                   JFrame.EXIT_ON_CLOSE);
24             frame.setTitle("JAASTest");
25             frame.setVisible(true);
26          });
27       }
28 }
```

Листинг 10.14. Исходный код из файла **jaas/JAASTest.policy**

```
1  grant codebase "file:login.jar"
2  {
3     permission java.awt.AWTPermission
```

```
4              "showWindowWithoutWarningBanner";
5    permission java.awt.AWTPermission "accessEventQueue";
6    permission javax.security.auth.AuthPermission
7              "createLoginContext.Login1";
8    permission javax.security.auth.AuthPermission
9              "doAsPrivileged";
10   permission javax.security.auth.AuthPermission
11             "modifyPrincipals";
12   permission java.io.FilePermission
13             "jaas/password.txt", "read";
14 };
15
16 grant principal jaas.SimplePrincipal "role=admin"
17 {
18   permission java.util.PropertyPermission "*", "read";
19 };
```

Листинг 10.15. Исходный код из файла **jaas/jaas.config**

```
1 Login1
2 {
3   jaas.SimpleLoginModule required pwfile =
4                          "jaas/password.txt" debug=true;
5 };
```

javax.security.auth.callback.CallbackHandler 1.4

- **void handle(Callback[] *callbacks*)**

  Обрабатывает указанные объекты обратного вызова, взаимодействуя при необходимости с пользователем, а затем сохраняет в них информацию, требующуюся для соблюдения безопасности.

javax.security.auth.callback.NameCallback 1.4

- **NameCallback(String *prompt*)**
- **NameCallback(String *prompt*, String *defaultName*)**

  Создают объект типа **NameCallback** с указанным приглашением и именем по умолчанию.

- **String getName()**
- **void setName(String *name*)**

  Получают или устанавливают имя, приобретаемое с помощью текущего объекта обратного вызова.

- **String getPrompt()**

  Получает приглашение, которое должно использоваться при запрашивании данного имени.

- **String getDefaultName()**

  Получает имя по умолчанию, которое должно использоваться при запрашивании данного имени.

javax.security.auth.callback.PasswordCallback 1.4

- **PasswordCallback(String *prompt*, boolean *echoOn*)**

  Создает объект типа **PasswordCallback** с указанным приглашением и признаком режима отображения эхо-символов.
- **char[] getPassword()**
- **void setPassword(char[] *password*)**

  Получают или устанавливают пароль, получаемый с помощью текущего объекта обратного вызова.
- **String getPrompt()**

  Получает приглашение, которое должно использоваться при запрашивании данного пароля.
- **boolean isEchoOn()**

  Получает признак режима отображения эхо-символов, который должен использоваться при запрашивании данного пароля.

javax.security.auth.spi.LoginModule 1.4

- **void initialize(Subject *subject*, CallbackHandler *handler*, Map<String,?> *sharedState*, Map<String,?> *options*)**

  Инициализирует объект типа **LoginModule** для аутентификации указанного субъекта. Для получения учетных данных использует указанный обработчик. Отображение ***sharedState*** применяется для взаимодействия с другими модулями регистрации, а отображение ***options*** — для хранения имен и значений, указанных при настройке данного экземпляра модуля регистрации.
- **boolean login()**

  Обеспечивает выполнение процесса регистрации и заполняет принципалы субъекта. Возвращает логическое значение **true** при удачном исходе регистрации.
- **boolean commit()**

  Если сценарий регистрации предполагает завершение в два этапа, то данный метод вызывается после удачного завершения всех модулей регистрации. Возвращает логическое значение **true** при удачном исходе операции.
- **boolean abort()**

  Вызывается, если неудачный исход выполнения другого модуля предполагает отказ от регистрации. Возвращает логическое значение **true** при удачном исходе операции.
- **boolean logout()**

  Отменяет регистрацию субъекта. Возвращает логическое значение **true** при удачном исходе операции.

10.4. Цифровые подписи

Как упоминалось ранее, интерес к платформе Java зародился с аплетов. Но программисты сразу поняли, что с практической точки зрения, несмотря на широкие анимационные возможности, аплеты не полностью поддерживают модель

безопасности. В версии JDK 1.0 аплеты очень строго контролировались. С другой стороны, в защищенной корпоративной сети компании риск атаки через аплеты минимален, поэтому логично было бы предоставить аплетам, выполняющимся в такой сети, дополнительные права. Разработчики из компании Sun Microsystems быстро осознали преимущества, которые дает применение аплетов, и поэтому решили предоставить пользователям возможность присваивать аплетам *разные* уровни защиты в зависимости от их происхождения. Так, если аплет поступает от надежного, заслуживающего доверия поставщика, то ему можно предоставить более обширные права доступа.

Для предоставления аплету дополнительных полномочий необходимо знать ответы на следующие вопросы.

1. Откуда поступил аплет?
2. Не был ли его код поврежден во время передачи?

За последние пятьдесят лет специалисты в области математики и информатики разработали немало сложных алгоритмов поддержки целостности данных и цифровых подписей. Многие из них реализованы в пакете `java.security`. Для применения таких алгоритмов совсем не обязательно понимать математические принципы, положенные в их основу. В последующих разделах описывается механизм свертки сообщений, позволяющий обнаружить факт изменений в документе, а также показывается, каким образом цифровая подпись способна подтверждать личность подписавшегося.

10.4.1. Свертки сообщений

Свертка сообщения — это цифровой "отпечаток" блока данных. Например, алгоритм безопасного хеширования SHA-1 (Secure Hash Algorithm #1) уплотняет любой блок данных в последовательность из 160 бит (20 байт). По аналогии с отпечатками пальцев, считается, что не существует двух одинаковых цифровых отпечатков по алгоритму SHA-1. На самом деле это не так, поскольку алгоритм SHA-1 поддерживает только 2^{160} отпечатков. Следовательно, теоретически они могут совпасть. Число 2^{160} настолько велико, что вероятность дублирования цифровых отпечатков очень мала, но насколько? Как утверждается в книге *True Odds: How risks Affect Your Everyday Life* Джеймса Уолша (James Walsh; издательство Merritt Publishing, 1996 г.), вероятность смерти от удара молнии составляет 1/30000. Если подсчитать вероятность такого же исхода для 10 человек (выбранных, например, злобным врагом), она окажется гораздо выше, чем вероятность наличия двух одинаковых цифровых отпечатков по алгоритму SHA-1. (Конечно, от удара молнии погибло гораздо больше, чем 10 человек, но здесь речь идет о специально выбранной группе людей.)

Свертка сообщения обладает двумя важными свойствами.

- Если изменяется один или несколько битов данных, то изменяется и свертка сообщения.
- Исходное сообщение нельзя изменить таким образом, чтобы полученное поддельное сообщение имело такую же свертку, как и у исходного сообщения.

Второе свойство, конечно, соблюдается с определенной степенью вероятности. Допустим, некий миллиардер составил следующее завещание:

"После смерти мое имущество должно быть разделено между моими детьми, но мой сын Джордж ничего не получит".

Отпечаток данного сообщения по алгоритму SHA-1 имеет такой вид:

```
12 5F 09 03 E7 31 30 19 2E A6 E7 E4 90 43 84 B4 38 99 8F 67
```

Допустим, недоверчивый миллиардер отдал текст завещания одному адвокату, а его отпечаток — другому. Допустим далее, что его сын Джордж подкупил адвоката, у которого хранится завещание, чтобы заменить слово **George** на слово **Bill**. В этом случае цифровой отпечаток по алгоритму SHA-1 поддельного завещания будет отличаться от аналогичного отпечатка исходного завещания:

```
7D F6 AB 08 EB 40 EC CD AB 74 ED E9 86 F9 ED 99 D1 45 B1 57
```

Может ли Джордж составить такое поддельное завещание, которое имело бы отпечаток оригинального завещания? Не может, потому что для перебора всех вариантов ему не хватило бы времени существования Земли, даже если бы он был счастливым обладателем миллиарда компьютеров, которые способны перебирать миллион вариантов завещания в секунду.

Для вычисления таких сверток сообщений был разработан целый ряд алгоритмов. Двумя наиболее известными из них являются алгоритм SHA-1, разработанный в США Национальным институтом стандартов и технологий (National Institute of Standards and Technology — NIST), и алгоритм MD5, изобретенный Рональдом Райвестом (Ronald Rivest) из Массачусетсского технологического института (Massachusetts Institute of Technologies — MIT). Оба эти алгоритма способны шифровать фрагменты сообщений своим оригинальным способом. Подробнее с ними можно ознакомиться, например, в книге *Cryptography and Network Security, 7th Edition* Вильяма Столлингса (William Stallings; издательство Prentice Hall, 2017 г.). Однако недавно в обоих алгоритмах были обнаружены незначительные изъяны, поэтому специалисты по шифрованию из института NIST рекомендуют пользоваться более надежными алгоритмами, в том числе SHA-256, SHA-384 или SHA-512.

Класс `MessageDigest` служит *фабрикой* для создания объектов, инкапсулирующих алгоритмы получения цифровых отпечатков. Этот класс содержит статический метод `getInstance()`, возвращающий экземпляр подкласса, производного от данного класса. Поэтому класс `MessageDigest` может выступать в роли фабричного класса или суперкласса для всех алгоритмов получения свертки сообщения. В приведенном ниже примере показано, каким образом получается объект для вычисления цифровых отпечатков по алгоритму SHA-1.

```
MessageDigest alg = MessageDigest.getInstance("SHA-1");
```

После создания объекта типа `MessageDigest` ему нужно передать все байты сообщения, повторно вызывая метод `update()`. Так, в приведенном ниже фрагменте кода содержимое файла передается полученному выше объекту `alg`, формирующему цифровой отпечаток.

```
InputStream in = . . .
int ch;
while ((ch = in.read()) != -1)
  alg.update((byte) ch);
```

С другой стороны, если байты размещаются в массиве, метод `update()` можно применить ко всему массиву следующим образом:

```
byte[] bytes = . . .;
alg.update(bytes);
```

Далее следует вызвать метод `digest()`, который дополняет вводимые данные недостающими битами (это необходимое условие для алгоритма получения цифровых отпечатков), вычисляет цифровой отпечаток и возвращает свертку сообщения в виде следующего байтового массива:

```
byte[] hash = alg.digest();
```

В примере программы из листинга 10.16 демонстрируется вычисление свертки сообщения. Чтобы запустить эту программу на выполнение, достаточно ввести следующую команду:

```
java hash.Digest hash/input.txt SHA-1
```

Если же аргументы данной программы не будут указаны в командной строке, то будет выдано приглашение ввести имя файла и наименование алгоритма для вычисления свертки сообщения.

Листинг 10.16. Исходный код из файла **`hash/Digest.java`**

```
1  package hash;
2
3  import java.io.*;
4  import java.nio.file.*;
5  import java.security.*;
6  import java.util.*;
7
8  /**
9   * В этой программе вычисляется свертка
10  * сообщения из файла
11  * @version 1.21 2018-04-10
12  * @author Cay Horstmann
13  */
14 public class Digest
15 {
16   /**
17    * @param args args[0] - имя файла,
18    *             args[1] - дополнительно алгоритм
19    *             (SHA-1, SHA-256 или MD5)
20    */
21   public static void main(String[] args)
22         throws IOException, GeneralSecurityException
23   {
24     var in = new Scanner(System.in);
25     String filename;
26     if (args.length >= 1)
27       filename = args[0];
28     else
```

```
29      {
30        System.out.print("File name: ");
31        filename = in.nextLine();
32      }
34      String algname;
35      if (args.length >= 2)
36        algname = args[1];
37      else
38      {
39        System.out.println("Select one of the following algorithms: ");
40        for (Provider p : Security.getProviders())
41          for (Provider.Service s : p.getServices())
42            if (s.getType().equals("MessageDigest"))
43              System.out.println(s.getAlgorithm());
44          System.out.print("Algorithm: ");
45          algname = in.nextLine();
46      }
47      MessageDigest alg = MessageDigest.getInstance(algname);
48      byte[] input = Files.readAllBytes(Paths.get(filename));
49      byte[] hash = alg.digest(input);
50      for (int i = 0; i < hash.length; i++)
51        System.out.printf("%02X ", hash[i] & 0xFF);
52      System.out.println();
53    }
54 }
```

java.security.MessageDigest 1.1

- **static MessageDigest getInstance(String *algorithmName*)**

  Возвращает объект типа **MessageDigest**, реализующий указанный алгоритм. Если такой алгоритм не поддерживается, то генерируется исключение типа **NoSuchAlgorithmExcetion**.

- **void update(byte *input*)**
- **void update(byte[]*input*)**
- **void update(byte[] *input*, int *offset*, int *len*)**

  Обновляют свертку сообщения, используя указанные байты.

- **byte[] digest()**

  Вычисляет свертку сообщения по алгоритму хеширования, возвращает вычисленную свертку и устанавливает объект алгоритма в исходное состояние.

- **void reset()**

  Устанавливает объект свертки сообщения в исходное состояние.

10.4.2. Подписание сообщений

В предыдущем разделе было показано, как создавать свертку сообщения, т.е. делать своего рода дактилоскопический отпечаток исходного сообщения. При изменении сообщения цифровой отпечаток измененного сообщения не будет совпадать с отпечатком исходного сообщения. Это означает, что при доставке

сообщения вместе с его цифровым отпечатком получатель сможет проверить, не было ли оно подделано. Но если злоумышленнику удастся перехватить и сообщение, и его исходный отпечаток, то он сможет легко изменить сообщение и переделать этот отпечаток так, как ему нужно. В конце концов, алгоритмы получения сверток сообщений всем известны и не требуют использования секретных ключей. В таком случае получатель поддельного сообщения и переделанного цифрового отпечатка так и не узнает, что его сообщение было изменено. Этот недостаток позволяют устранить цифровые подписи.

Чтобы стал понятнее принцип действия цифровых подписей, придется сначала ввести некоторые понятия из области *шифрования открытым ключом*. Основными в этой области являются такие понятия, как *открытый ключ* и *секретный ключ*. Открытый ключ сообщается всем, а секретный держится в строгом секрете. Соответствие между этими ключами устанавливается на основании математических отношений, которые здесь не рассматриваются. (Тем, кому интересно узнать о них более подробно, рекомендуется книга *The Handbook of Applied Cryptography*, оперативно доступная для заказа по адресу `http://www.cacr.math.uwaterloo.ca/hac/`.)

Эти ключи довольно длинные и сложные. В качестве примера ниже приведена пара открытого и секретного ключей, полученных с помощью алгоритма DSA (Digital Signature Algorithm — алгоритм создания цифровых подписей).

Открытый ключ:

```
: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f 3ae1617a
e01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee7 37592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d 14271b9
e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be 794ca4

y: c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdab9ca2 d2a8123
ce5a8018b8161a760480fadd040b927281ddb22cb9bc4df596d7de4d1b9 77d50
```

Соответствующий ему секретный ключ:

```
p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3 ae1617a
e01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee737 592e17

q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5

g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d 14271b9
e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be 794ca4

x: 146c09f881656cc6c51f27ea6c3a91b85ed1d70a
```

Считается, что вывести один ключ из другого практически невозможно. Несмотря на то что открытый ключ известен всем, узнать с его помощью секретный ключ злоумышленникам никогда не удастся, сколько бы вычислительных ресурсов ни было в их распоряжении.

В то, что никому не удастся вычислить секретный ключ на основе открытого ключа, трудно поверить, но, по крайней мере, пока еще никто не изобрел алгоритм, способный делать нечто подобное с ключами, генерируемыми с помощью

самых распространенных в настоящее время алгоритмов шифрования. Из-за того что эти ключи достаточно длинные, для их расшифровки методом "грубой силы", т.е. путем простого перебора всех возможных вариантов, потребуется больше компьютеров, чем может быть вообще создано из всех атомов солнечной системы, и не одна тысяча лет. Разумеется, не исключено, что кому-нибудь удастся придумать для разгадывания ключей шифрования более эффективный алгоритм, чем простой перебор возможных вариантов.

Рассмотрим в качестве примера алгоритм шифрования RSA, изобретенный Райвестом (Rivest), Шамиром (Shamir) и Адлеманом (Adleman). Принцип действия этого алгоритма основывается на сложности разложения больших чисел на простые множители. За последние двадцать лет многие известные математики пытались разработать удачные алгоритмы разложения чисел на простые множители, но добиться этого пока еще никому не удалось. Из-за этого большинство специалистов по шифрованию считают, что в настоящее время ключи длиной 2000 битов являются абсолютно защищенными от любых попыток взлома. Алгоритм шифрования DSA также считается довольно надежным. На рис. 10.9 показано, как выглядит процесс шифрования по алгоритму DSA на практике.

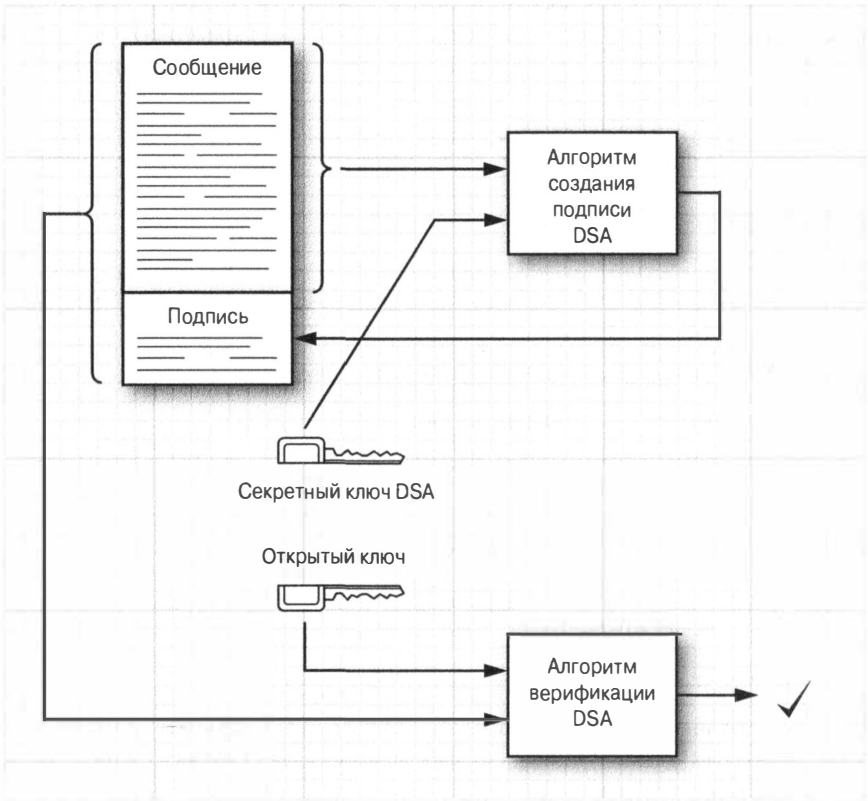


Рис. 10.9. Обмен сообщениями с использованием цифровой подписи, открытых ключей и алгоритма шифрования DSA

Допустим, Алиса хочет послать сообщение Бобу, а Боб желает быть уверенным в том, что сообщение поступило именно от Алисы, а не от какого-то самозванца. Для этого Алиса составляет сообщение и *подписывает* свертку этого сообщения с помощью своего секретного ключа. Далее Боб получает копию ее открытого ключа и применяет его для *верификации* подписи. При удачном завершении процесса верификации Боб может быть уверен в следующем:

- исходное сообщение не было изменено;
- сообщение было подписано Алисой, обладающей секретным ключом, соответствующим тому открытому ключу, который Боб использовал для верификации.

Из этого примера ясно видно, почему так важны секретные ключи. Если кто-нибудь выкрадет секретный ключ Алисы или если правительство попросит ее расшифровать его, ей грозят неприятности, потому что выкравший ключ злоумышленник или представитель правительства сможет легко подделывать ее сообщения, заявки на пересылку денежных средств и выполнять другие подобные операции, а все будут считать, что все это делает сама Алиса.

10.4.3. Верификация подписи

В состав комплекта JDK входит утилита `keytool` командной строки, предназначенная для генерирования сертификатов и управления ими. Пока что она способна работать только в режиме командной строки, но можно надеяться, что в последующих выпусках JDK ее функциональные возможности будут доступны и в виде других, более удобных для пользователей версиях. В этом разделе она используется для того, чтобы показать, каким образом Алиса может подписывать документ и отправлять его Бобу, а Боб — проверять и удостоверяться в том, что документ был действительно подписан Алисой, а не каким-нибудь самозванцем.

Утилита `keytool` позволяет управлять *хранилищами ключей*, базами данных сертификатов и парами секретных и открытых ключей. У каждой записи в хранилище ключей имеется свой *псевдоним*. Ниже показано, каким образом Алиса может создать хранилище `alice.store` и сгенерировать пару ключей с псевдонимом `alice`, воспользовавшись утилитой `keytool`.

```
keytool -genkeypair -keystore alice.certs -alias alice
```

При создании или открытии хранилища появляется приглашение ввести пароль. В данном примере вводится простое слово **`secret`**. Но при создании хранилища ключей с помощью утилиты `keytool` для каких-нибудь более серьезных целей рекомендуется выбрать надежный пароль и хранить его в полном секрете. При генерировании ключа появляется приглашение ввести следующую информацию:

```
Enter keystore password: secret
(Введите пароль для хранилища ключей)
Reenter new password: secret
(Еще раз введите новый пароль)
What is your first and last name?
(Ваше имя и фамилия [неизвестно])
[Unknown]: Alice Lee
```

```
What is the name of your organizational unit?
(Название подразделения вашей организации [неизвестно])
[Unknown]: Engineering Department
What is the name of your organization?
(Название вашей организации [неизвестно])
[Unknown]: ACME Software
What is the name of your City or Locality?
(Название вашего города или местности [неизвестно])
[Unknown]: San Francisco
What is the name of your State or Province?
(Название вашего штата или провинции [неизвестно])
[Unknown]: CA
What is the two-letter country code for this unit?
(Введите двухбуквенный код страны для данного
 подразделения [неизвестно])
[Unknown]: US
Is <CN=Alice Lee, OU=Engineering Department, O=ACME Software,
    L=San Francisco, ST=CA, C=US> correct?
(Все правильно?)
[no]: yes
```

В утилите `keytool` для идентификации владельцев ключей и создателей сертификатов используются имена по стандарту X.500 с составляющими для указания общего имени (Common Name — CM), организационного подразделения (Organizational Unit — OU), организации (Organization — O), местонахождения (Location — L), штата (State — ST) и страны (Country — C). И в завершение требуется указать пароль для ключа или нажать клавишу <Enter>, чтобы использовать его в качестве пароля для хранилища ключей.

Допустим, Алисе требуется предоставить Бобу копию своего открытого ключа. В таком случае ей нужно сначала экспортировать файл сертификата следующим образом:

```
keytool -exportcert -keystore alice.certs \
        -alias alice -file alice.cer
```

Теперь она может отправить этот сертификат Бобу. Боб же, получив этот сертификат, может распечатать его по следующей команде:

```
keytool -printcert -file alice.cer
```

Ниже приведен пример такой распечатки. Если Боб захочет проверить, тот ли сертификат он получил, он может позвонить Алисе и сверить свою копию с исходным цифровым отпечатком сертификата по телефону.

```
Owner: CN=Alice Lee, OU=Engineering Department,
       O=ACME Software, L=San Francisco, ST=CA, C=US
Issuer: CN=Alice Lee, OU=Engineering Department,
         O=ACME Software, L=San Francisco, ST=CA, C=US
Serial number: 470835ce
Valid from: Sat Oct 06 18:26:38 PDT 2007 until:
            Fri Jan 04 17:26:38 PST 2008
Certificate fingerprints:
    MD5: BC:18:15:27:85:69:48:B1:5A:C3:0B:1C:C6:11:B7:81
    SHA1: 31:0A:A0:B8:C2:8B:3B:B6:85:7C:EF:C0:57:E5:94:
          95:61:47:6D:34
    Signature algorithm name: SHA1withDSA
    Version: 3
```

НА ЗАМЕТКУ! Некоторые создатели сертификатов публикуют цифровые отпечатки своих сертификатов на веб-сайтах. Например, чтобы проверить сертификат компании DigiCert из каталога хранилища ключей ***jre/lib/security/cacerts***, можно отобразить сначала содержимое этого каталога по следующей команде с параметром **-list**:

```
keytool -list -v -keystore jre/lib/security/cacerts
```

Паролем для доступа к данному хранилищу является слово **changeit**. А один из сертификатов в нем будет выглядеть так:

```
Owner: CN=DigiCert Assured ID Root G3, OU=www.digicert.com,
        O=DigiCert Inc, C=US
Issuer: CN=DigiCert Assured ID Root G3, OU=www.digicert.com,
         O=DigiCert Inc, C=US
Serial number: ba15afa1ddfa0b54944afcd24a06cec
Valid from: Thu Aug 01 14:00:00 CEST 2013 until:
      Fri Jan 15 13:00:00 CET 2038
Certificate fingerprints:
SHA1: F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:
      AB:30:89
SHA256: 7E:37:CB:8B:4C:47:09:0C:AB:36:55:1B:A6:F4:5D:B8:
        40:68:0F:BA:
16:6A:95:2D:B1:00:71:7F:43:05:3F:C2
```

Чтобы удостовериться, что данный сертификат является действительным, достаточно посетить веб-сайт компании DigiCert по адресу **https://www.digicert.com/digicert-root-certificates.htm**.

Убедившись в правильности сертификата, Боб может далее импортировать его в свое хранилище ключей по следующей команде:

```
keytool -importcert -keystore bob.certs -alias alice \
        -file alice.cer
```

ВНИМАНИЕ! Ни в коем случае не следует импортировать непроверенный сертификат в хранилище ключей. Ведь после ввода сертификата в хранилище ключей любая программа, пользующаяся хранилищем ключей, предполагает, что данный сертификат можно использовать для проверки цифровых подписей.

После этого Алиса может отправлять Бобу подписанные документы. Подписывать и проверять архивные JAR-файлы можно с помощью утилиты `jarsigner`. Сначала Алисе следует ввести подписываемый документ в архивный JAR-файл:

```
jar cvf document.jar document.txt
```

Затем с помощью утилиты `jarsigner` она может добавить к этому файлу свою цифровую подпись. Для этого ей достаточно указать хранилище ключей, архивный JAR-файл и псевдоним используемого ключа в следующей команде:

```
jarsigner -keystore alice.certs document.jar alice
```

Получив этот файл, Боб может воспользоваться для его верификации утилитой `jarsigner` с параметром `-verify`, как показано ниже.

```
jarsigner -verify -keystore bob.certs document.jar
```

Указывать псевдоним ключа при этом Бобу не нужно. Утилита `jarsigner` сама отыщет в цифровой подписи указанное в формате X.500 имя владельца ключа и произведет поиск соответствующих ему сертификатов в хранилище ключей.

Если архивный JAR-файл не поврежден и цифровая подпись совпадает, тогда утилита `jarsigner` отобразит приведенное ниже сообщение. В противном случае вместо него появится сообщение об ошибке:

```
jar verified.
```

10.4.4. Проблема аутентификации

Допустим, вы получаете сообщение от своей подруги Алисы, которая подписала его описанным выше способом, используя свой секретный ключ. У вас уже может иметься копия ее открытого ключа, а иначе вам нетрудно будет получить ее у самой Алисы или на ее веб-странице. Наличие этого ключа позволяет легко удостовериться, что данное сообщение действительно написала Алиса и что оно не было кем-то подделано. А теперь допустим, что вы получаете сообщение от какого-то незнакомца, который представляется сотрудником известной компании по разработке программного обеспечения и предлагает вам воспользоваться программой, прикрепленной им к данному сообщению. Этот незнакомец даже присылает вам копию своего открытого ключа, чтобы вы могли удостовериться, что именно он является автором данного сообщения. Вы проверяете его и убеждаетесь, что подпись действительна. Это подтверждает лишь то, что сообщение было подписано с помощью соответствующего секретного ключа и что оно не было подделано.

Но имейте в виду, что *о самом авторе сообщения вам все равно ничего не известно*! Сгенерировать секретный и открытый ключи, подписать сообщение с помощью секретного ключа и отправить его вместе с подходящим открытым ключом мог кто угодно. Выяснение личности отправителя называется *проблемой аутентификации*.

Обычно проблема аутентификации разрешается очень просто. Например, у вас и у отправителя может быть общий знакомый, которому вы оба доверяете. Тогда отправитель может встретиться с вашим знакомым лично и передать ему диск с открытым ключом. Затем ваш знакомый может встретиться с вами, подтвердить, что он действительно знает отправителя и что тот действительно работает на известную компанию по разработке программного обеспечения, после чего передать вам диск с открытым ключом (рис. 10.10). В таком случае получается, что гарантом аутентичности отправителя будет ваш знакомый.

На самом деле вашему знакомому совсем не обязательно встречаться с вами. Он может просто подписать файл с открытым ключом незнакомца с помощью своего секретного ключа и послать его вам (рис. 10.11). При получении файла с открытым ключом вы можете проверить подпись своего друга. А поскольку вы ему доверяете, то у вас не сомнений, что он проверил учетные данные незнакомца, прежде чем добавлять свою цифровую подпись.

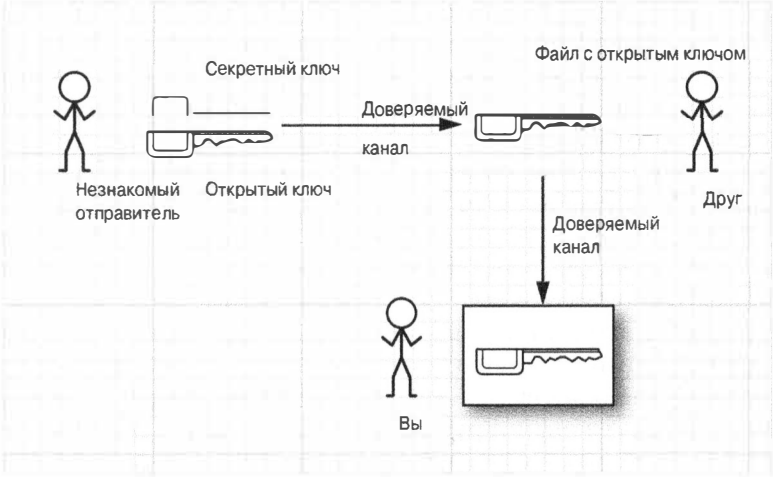


Рис. 10.10. Аутентификация через доверенного посредника

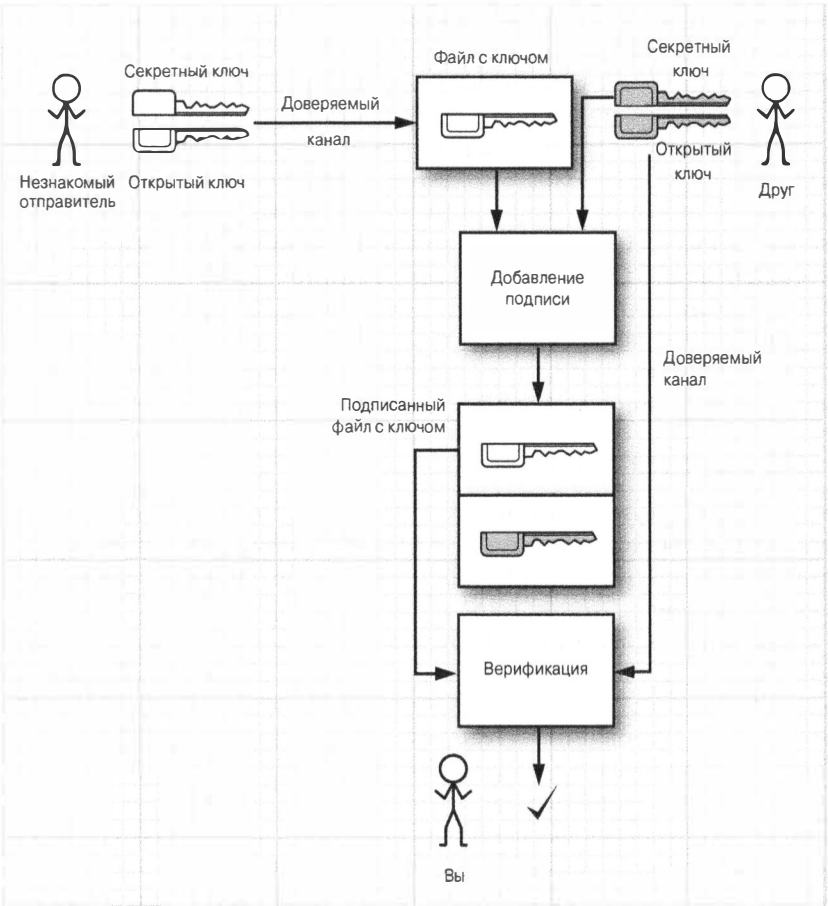


Рис. 10.11. Аутентификация через подпись доверенного посредника

Но у вас может и не быть такого общего знакомого. В некоторых моделях доверительных отношений предполагается, что всегда существует некая "цепочка доверия", т.е. цепочка общих знакомых, всем звеньям которой можно доверять. Но на практике так бывает далеко не всегда. Вы можете доверять своей подруге Алисе и знать, что она доверяет Бобу, но сами-то вы Боба не знаете, а следовательно, не до конца уверены, стоит ли ему доверять. Другая модель доверительных отношений предполагает наличие какого-то одного хорошо зарекомендовавшего себя человека или организации, которой все могут безоговорочно доверять. Наиболее известными примерами такой организации служат компании DigiCert, GlobalSign и Entrust.

Вам часто будут встречаться цифровые подписи, заверенные одной или несколькими ручающимися за аутентичность организациями, и поэтому вам придется самостоятельно оценивать, насколько им стоит доверять. Вы можете доверять какому-нибудь сертифицирующему органу, например, потому, что часто встречаете его логотип на многих веб-страницах, или потому, что слышали, будто бы в нем применяются невероятные меры безопасности.

Но вы все равно должны иметь ясное представление, каким образом в подобных организациях производится аутентификация и что именно они проверяют. Получить идентификатор "класса 1" от сертифицирующего органа можно, заполнив веб-форму и заплатив небольшой взнос. В этой форме следует указать имя, организацию, страну и адрес электронной почты. После заполнения формы по указанному адресу отсылается ключ или инструкции, как его получить. Таким образом, уверенным можно быть только в подлинности электронного адреса, потому что имя и название организации не проверяются и могут быть любыми. Существуют и более строгие классы идентификаторов, например, при желании получить идентификатор "класса 3" сертифицирующий орган потребует представить отчет о финансовом положении, официально заверенный у нотариуса. У других организаций, занимающихся аутентификацией, будут другие требования. Поэтому очень важно, чтобы при получении заверенного цифровой подписью сообщения вы ясно понимали, каким образом проверялась его подлинность.

10.4.5. Подписание сертификатов

В разделе 10.4.3 было показано, как Алиса воспользовалась самостоятельно подписываемым сертификатом для передачи открытого ключа Бобу. Но ведь Бобу пришлось проверить подлинность этого сертификата, сверив его с исходным цифровым отпечатком от Алисы.

Допустим теперь, что Алисе требуется послать подписанное сообщение своей коллеге Синди, а та не любит тратить время на сверку цифровых подписей с исходными отпечатками. В этом случае Синди может помочь посредническая организация, которой она могла бы доверить проверку всех цифровых подписей. В рассматриваемом здесь примере предполагается, что такой организацией является отдел информационных ресурсов компании ACME Software.

Этот отдел занимается выдачей сертификатов, т.е. выполняет функции *сертифицирующего органа*. У каждого сотрудника компании ACME Software имеется полученный из этого отдела открытый ключ, который находится в хранилище ключей и перед установкой был тщательно сверен с цифровым отпечатком

системным администратором. Этот сертифицирующий орган занимается также подписанием ключей всех сотрудников компании ACME Software. Благодаря этому хранилище ключей автоматически доверяет устанавливаемым ключам сотрудников данной компании, поскольку они подписаны с помощью доверяемого ключа.

Ниже поясняется, как сымитировать этот процесс. Сначала создается хранилище ключей `acmesoft.certs`. Затем генерируется пара ключей и экспортируется открытый ключ по следующим командам:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
keytool -exportcert -keystore acmesoft.certs \
        -alias acmeroot -file acmeroot.cer
```

Открытый ключ экспортируется в "самозаверенный" сертификат, а затем вводится в хранилище ключей каждого сотрудника следующим образом:

```
keytool -importcert -keystore cindy.certs -alias acmeroot \
        -file acmeroot.cer
```

Чтобы иметь возможность отправлять подписанные сообщения Синди и любым другим сотрудникам компании ACME Software, Алисе нужно принести свой сертификат в отдел информационных ресурсов данной компании и подписать его. Утилита `keytool`, к сожалению, не предоставляет никаких средств для решения подобной задачи. Для восполнения этого пробела в исходном коде, прилагаемом к данной книге, предоставляется класс `CertificateSigner`. В этом случае уполномоченному сотруднику компании ACME Software остается только проверить личность Алисы и сформировать подписанный сертификат по следующей команде:

```
java CertificateSigner -keystore acmesoft.certs \
     -alias acmeroot -infile alice.cer \
     -outfile alice_signedby_acmeroot.cer
```

Очевидно, что выполнение такой команды связано с известным риском и требует, чтобы у применяемой для подписания сертификата программы был доступ к хранилищу ключей компании ACME Software, а у уполномоченного сотрудника — необходимый для этого пароль.

После этого Алиса может передать файл `alice_signedby_acmeroot.cer` Синди и любому другому сотруднику компании ACME Software. В качестве альтернативного варианта этот файл может быть также сохранен в каталоге данной компании. Напомним, что в этом файле содержится открытый ключ Алисы и подтверждение компании ACME Software, что данный ключ действительно принадлежит Алисе. Далее Синди может импортировать подписанный сертификат Алисы в свое хранилище ключей по следующей команде:

```
keytool -importcert -keystore cindy.certs -alias alice \
        -file alice_signedby_acmeroot.cer
```

При выполнении этой команды в хранилище ключей будет автоматически проверено, был ли данный сертификат подписан с помощью уже имеющегося в нем доверяемого корневого ключа. Благодаря этому Синди *не* придется сверять его с исходным цифровым отпечатком данного сертификата. Введя один раз корневой сертификат и сертификаты всех остальных лиц, часто присылающих ей

документы, Синди избавляется от необходимости впредь беспокоиться о хранилище ключей.

10.4.6. Запросы сертификатов

В предыдущем разделе был рассмотрен пример имитации сертифицирующего органа с помощью хранилища ключей и класса `CertificateSigner`. Но в большинстве сертифицирующих органов для управления сертификатами применяется более сложное программное обеспечение и несколько иные форматы сертификатов. В этом разделе обсуждаются дополнительные действия, которые требуется выполнить для организации нормального взаимодействия с подобным программным обеспечением.

Выберем для примера пакет программного обеспечения OpenSSL. Это программное обеспечение, как правило, предварительно устанавливается во многих системах Linux и Mac OS X вместе с портом Cygwin. Если оно не установлено, его можно без особого труда загрузить по адресу `https://www.openssl.org`.

Чтобы создать сертифицирующий орган, следует запустить сценарий `CA`. Точное место расположения этого сценария зависит от конкретной операционной системы. Так, в ОС Ubuntu его можно запустить по следующей команде:

```
/usr/lib/ssl/misc/CA.pl -newca
```

Этот сценарий создаст в текущем каталоге подкаталог `demoCA`, содержащий пару корневых ключей и хранилище для сертификатов и списков аннулирования сертификатов.

Далее необходимо импортировать открытый ключ в хранилище ключей, созданное на Java для всех сотрудников. Но из-за того что этот ключ имеет формат PEM (Privacy Enhanced Mail — почта повышенной секретности), а не легко распознаваемый хранилищем ключей формат DER, сначала придется скопировать файл `demoCA/cacert.pem` в файл `acmeroot.pem`, открыть его в текстовом редакторе и удалить все, что находится перед строкой `-----BEGIN CERTIFICATE-----` и после строки `-----END CERTIFICATE-----`.

После этого файл `acmeroot.pem` можно скопировать в каждое хранилище ключей обычным способом, как показано ниже. Невероятно, но факт: утилита `keytool` не способна выполнить эту операцию редактирования самостоятельно.

```
keytool -importcert -keystore cindy.certs -alias alice \
        -file acmeroot.pem
```

Чтобы подписать открытый ключ Алисы, необходимо сначала сформировать *запрос сертификата* формате PEM по следующей команде:

```
keytool -certreq -keystore alice.store -alias alice \
        -file alice.pem
```

Далее для подписания этого сертификата необходимо выполнить следующую команду:

```
openssl ca -in alice.pem -out alice_signedby_acmeroot.pem
```

После этого, как и прежде, из файла `alice_signedby_acmeroot.pem` следует удалить все строки, которые находятся за пределами маркеров `BEGIN CERTIFICATE/END CERTIFICATE`, а затем импортировать этот файл в хранилище

ключей по приведенной ниже команде. Аналогичным образом можно подписать сертификат с помощью ключа, выданного сертифицирующим органом.

```
keytool -importcert -keystore cindy.certs -alias alice \
        -file alice_signedby_acmeroot.pem
```

10.4.7. Подписание кода

Технология аутентификации чаще всего применяется для подписания исполняемых программ. Копируя программу из сети, пользователь вполне обоснованно может потребовать гарантий ее подлинности, поскольку такая программа может нанести вред, если она заражена каким-нибудь вирусом. Поэтому требуется полная уверенность в том, что программа предоставлена надежным источником и во время пересылки не была изменена.

В этом разделе будет показано, как подписываются архивные JAR-файлы и настраиваются средства Java для проверки достоверности подписи. Такая возможность была предусмотрена для подключаемого модуля Java Plug-in, предназначенного для запуска аплетов и приложений Java Web Start. И хотя эти технологии не находят больше широкого применения, их, возможно, придется поддерживать в унаследованных программных продуктах.

В первоначальной версии Java аплеты допускалось выполнять в "песочнице" с ограниченными полномочиями непосредственно после их загрузки. Если же пользователям требовались аплеты, способные получать доступ к локальной файловой системе, устанавливать сетевые соединения и т.д., они должны были явным образом давать свое согласие на подобные операции. Чтобы гарантировать от намеренного повреждения кода аплета в процессе его выполнения, такой код снабжали цифровой подписью.

Обратимся к конкретному примеру. Допустим, что, выйдя в Интернет, пользователь открывает веб-страницу, на которой предлагается запустить прикладную программу неизвестного поставщика, как, например, показано на рис. 10.12. Такая программа подписана сертификатом разработчика программного обеспечения. В диалоговом окне указывается разработчик данной программы и создатель сертификата. Пользователь должен решить, следует ли санкционировать выполнение выбранной прикладной программы.

Попробуем выяснить, что известно пользователю в подобной ситуации и что может повлиять на его решение. А известно следующее.

- Компания Thawte продала сертификат разработчику программного обеспечения.
- Прикладная программа действительно подписана этим сертификатом и во время пересылки не была изменена.
- Сертификат действительно подписан компанией Thawte и проверен с помощью открытого ключа, который находится в локальном файле `cacerts`.

Безусловно, ничто из приведенного выше не свидетельствует о том, что данную программу можно выполнить безопасно. Можно ли доверять поставщику, если известно только его название и тот факт, что компания Thawte продала ему сертификат разработчика программного обеспечения? Этих сведений явно недостаточно для принятия взвешенного решения.

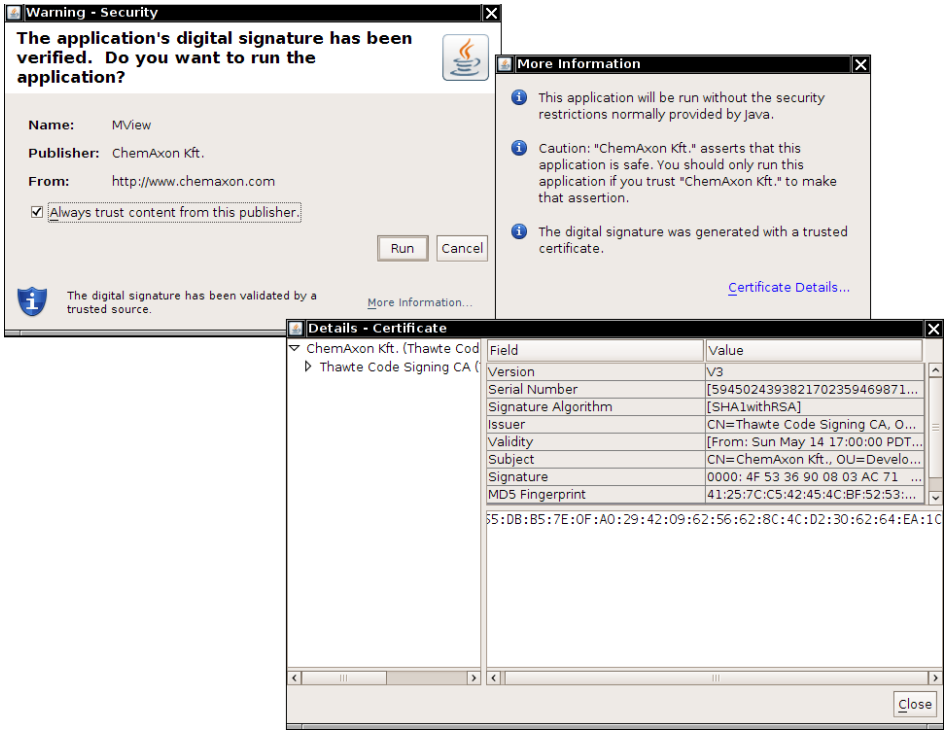


Рис. 10.12. Запуск подписанной прикладной программы

Сертификаты более пригодны для разработки внутренних корпоративных сетей, администраторы которых могут установить файлы правил защиты и сертификаты на локальных машинах, чтобы для запуска на выполнение доверяемого кода не требовалось вмешательство пользователей. Всякий раз, когда подключаемый модуль Java Plug-in загружает подписанный код, он обращается за правами доступа к файлу правил защиты, а к хранилищу ключей — за подписями.

Далее в этом разделе поясняется, каким образом создаются файлы правил защиты, дающие разрешение на выполнение прикладного кода из известных источников. Допустим, компании ACME Software требуется, чтобы ее сотрудники могли запускать определенные программы, требующие доступа к локальным файлам, но сделать так, чтобы эти программы были доступны через браузер в виде приложений Web Start.

Как было показано ранее в этой главе, компания ACME Software могла бы идентифицировать программы по их кодовой базе. Но это означает, что ей пришлось бы обновлять файлы прав защиты всякий раз, когда программы переносятся на другой сервер. Поэтому в компании ACME Software вместо этого было принято решение *подписывать* архивные JAR-файлы, содержащие программный код.

Для этого сначала формируется корневой сертификат по следующей команде:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
```

Разумеется, хранилище ключей, содержащее секретный корневой ключ, должно обязательно находиться в безопасном месте. Поэтому создается второе хранилище ключей `client.certs` для размещения в нем открытых сертификатов, и в него сразу же вводится открытый сертификат `acmeroot`:

```
keytool -exportcert -keystore acmesoft.certs \
        -alias acmeroot -file acmeroot.cer
keytool -importcert -keystore client.certs \
        -alias acmeroot -file acmeroot.cer
```

Далее уполномоченный сотрудник компании ACME Software может запустить утилиту `jarsigner`, чтобы подписать любую прикладную программу, указав архивный JAR-файл и псевдоним секретного ключа следующим образом:

```
jarsigner -keystore acmesoft.certs ACMEApp.jar acmeroot
```

После этого приложение Web Start можно считать готовым к развертыванию на веб-сервере. Теперь перейдем к настройке безопасности на стороне клиента. Файл правил защиты должен рассылаться каждой клиентской машине. Для ссылки на хранилище ключей файл правил защиты начинается со следующей строки:

```
keystore "URL_хранилища_ключей", "тип_хранилища_ключей";
```

Указанный URL может быть как абсолютным, так и относительным. Относительные URL являются таковыми по отношению к месту расположения файла правил защиты. Если хранилище формировалось с помощью утилиты `keytool`, оно будет относиться к типу `JKS`, как показано в приведенном ниже примере.

```
keystore "client.certs", "JKS";
```

Далее, операторы `grant` в файле правил защиты могут быть снабжены суффиксами `signedBy "псевдоним"`, как выделено ниже полужирным. В этих операторах предоставляются полномочия любому коду, который может быть проверен с помощью открытого ключа, связанного с указанным псевдонимом.

```
grant signedBy "acmeroot"
{
  . . .
};
```

10.5. Шифрование

До сих пор рассматривалась аутентификация с помощью цифровых подписей. Еще одним важным средством обеспечения безопасности является *шифрование*. Информация, заверенная цифровой подписью, доступна для просмотра, а подпись лишь подтверждает, что эта информация не была изменена. Для просмотра зашифрованных данных этого недостаточно, поскольку их нужно расшифровать с помощью согласованного ключа.

Аутентификации оказывается достаточно для подписания кода, который не нужно скрывать. А шифрование требуется в тех случаях, когда прикладные программы передают конфиденциальную информацию, например, номера кредитных карточек и прочие личные данные.

В прошлом во многих компаниях существовали патентные и экспортные ограничения на использование эффективных алгоритмов шифрования. Теперь же все эти ограничения стали, к счастью, менее жесткими, а срок патентных ограничений на использование ряда важных алгоритмов шифрования и вовсе истек. В стандартной библиотеке текущей версии Java предусмотрены превосходные средства шифрования.

10.5.1. Симметричные шифры

Криптографические расширения Java содержат класс `Cipher`, который является суперклассом для всех классов, имеющих отношение к шифрованию. Для создания объекта, реализующего алгоритм шифрования, метод `getInstance()` используется одним из следующих способов:

```
Cipher cipher = Cipher.getInstance(algorithName);
```

или

```
Cipher cipher = Cipher.getInstance(algorithName,
                                   providerName);
```

В комплекте JDK для всех шифров используется поставщик `SunJCE`. Если имя поставщика не указано явно, то по умолчанию принимается имя `SunJCE`. Если же требуется воспользоваться алгоритмами шифрования, которые не поддерживаются инструментальными средствами компании Oracle, следует указать другого поставщика. Название алгоритма шифрования задается в виде символьной строки, например `"DES"` или `"DES/CBC/PKCS5Padding"`.

Алгоритм DES (Data Encryption Standard — стандарт шифрования данных) — один из наиболее старых алгоритмов шифрования с длиной ключа 56 бит. В настоящее время он считается устаревшим, поскольку может взламываться методом "грубой силы". Намного более эффективным оказывается появившийся после него алгоритм AES (Advanced Encryption Standard — усовершенствованный стандарт шифрования), подробное описание которого можно найти по адресу `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf`. В рассматриваемых далее примерах будет использоваться алгоритм шифрования AES.

Как только будет создан объект, реализующий алгоритм шифрования, его необходимо инициализировать, указав режим и ключ шифрования, т.е. установив параметры `mode` и `key` следующим образом:

```
int mode = . . .;
Key key = . . .;
cipher.init(mode, key);
```

Параметр `mode` может принимать значение одной из следующих констант:

```
Cipher.ENCRYPT_MODE
Cipher.DECRYPT_MODE
Cipher.WRAP_MODE
Cipher.UNWRAP_MODE
```

Режимы *свертывания* и *развертывания* применяются для шифрования одного ключа на основе другого. Пример применения таких режимов будет приведен в следующем разделе. После этого можно повторно вызывать метод `update()` для шифрования всех требующихся блоков данных, как показано ниже.

```
int blockSize = cipher.getBlockSize();
var inBytes = new byte[blockSize];
. . . // прочитать байты из входного массива inBytes
int outputSize= cipher.getOutputSize(blockSize);
var outBytes = new byte[outputSize];
int outLength = cipher.update(inBytes, 0,
                              outputSize, outBytes);
. . . // записать байты в выходной массив outBytes
```

По завершении следует вызвать метод `doFinal()` один раз. Если доступен последний блок вводимых данных (меньшего объема, чем указано в переменной `blockSize`), то данный метод должен быть вызван следующим образом:

```
outBytes = cipher.doFinal(inBytes, 0, inLength);
```

Если были зашифрованы все вводимые данные, то данный метод должен быть вызван таким образом:

```
outBytes = cipher.doFinal();
```

Вызывать метод `doFinal()` требуется для того, чтобы заполнить завершающий блок данных. Рассмотрим, например, алгоритм шифрования DES, в котором размер блока составляет 8 байт. Допустим, что размер последнего блока вводимых данных оказывается меньше 8 байт. Разумеется, недостающие байты можно дополнить нулями таким образом, чтобы блок занимал 8 байт, а затем зашифровать его. Но после расшифровки блоков данных полученный результат будет содержать несколько присоединенных в конце нулей, а следовательно, будет несколько отличаться от исходного файла данных. Это может вызвать определенное затруднение, и для его преодоления как раз и требуется *схема заполнения*. Одной из наиболее часто применяемых является схема заполнения, описанная в документе Public Key Cryptography Standard #5 (PKCS — стандарт шифрования открытым ключом) специалистами из компании RSA Security, Inc. (`https://tools.ietf.org/html/rfc2898`).

В этой схеме последний блок заполняется не нулями, а числами, равными недостающему количеству байтов. Иными словами, если `L` — это последний (неполный) блок данных, то он будет заполнен следующим образом:

```
L 01                    если length(L) = 7
L 02 02                 если length(L) = 6
L 03 03 03              если length(L) = 5
...
L 07 07 07 07 07 07 07  если length(L) = 1
```

Наконец, если длина вводимых данных делится нацело на 8, то к вводимым данным присоединяется и затем шифруется только один такой блок, как показано ниже. При расшифровке на количество отбрасываемых заполняющих символов указывает самый последний байт простого текста.

```
08 08 08 08 08 08 08 08
```

10.5.2. Генерирование ключей шифрования

Для шифрования требуется сгенерировать ключ. Каждый алгоритм шифрования предусматривает свой формат для ключей, но самое главное, чтобы их генерирование выполнялось произвольным образом. Чтобы получить

сгенерированный совершенно произвольно ключ, требуется выполнить следующие действия.

1. Создать объект типа `KeyGenerator`.
2. Инициализировать генератор случайных чисел. Если блок шифра имеет переменную длину, необходимо также указать желаемую длину блока.
3. Вызвать метод `generateKey()`.

В качестве примера ниже показано, каким образом генерируется ключ шифрования по алгоритму AES.

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
var random = new SecureRandom(); // см. пояснение ниже
keygen.init(random);
Key key = keygen.generateKey();
```

С другой стороны, ключ шифрования можно также сгенерировать из фиксированного набора необработанных данных (например, пароля или вводимых с клавиатуры символов). В таком случае следует воспользоваться классом `SecretKeyFactory`, как показано ниже.

```
// 16 байт для ключа шифрования по алгоритму AES:
byte[] keyData = . . .;
var key = new SecretKeySpec(keyData, "AES");
```

При генерировании ключей следует удостовериться, что для этой цели используются *подлинно случайные* числа. Например, обычный генератор случайных чисел, реализуемый в классе `Random`, где в роли начальных значений выступают текущие дата и время, не будет генерировать случайные в достаточной степени числа. Допустим, системные часы показывают время с точностью до 1/10 секунды. Это означает, что на каждый день может приходиться максимум по 864000 начальных значений. Следовательно, узнав день генерирования ключа шифрования (его зачастую нетрудно вычислить по дате сообщения или дате истечения срока действия сертификата), злоумышленнику останется лишь сгенерировать все возможные значения за этот день.

Класс `SecureRandom` генерирует намного более надежные случайные числа, чем класс `Random`. Но предоставлять ему начальное значение, с которого должна начинаться числовая последовательность в произвольной точке, все равно придется. Для этого удобнее всего получить случайные данные, вводимые из какого-нибудь аппаратного устройства вроде генератора белого шума. С другой стороны, случайные данные можно получить, запросив у пользователя ввести любую бессмысленную последовательность символов, из которой в качестве начального случайного значения должен выбираться только один или два бита. После накопления таких случайных битов в массив байтов они должны быть переданы методу `setSeed()`, как показано ниже.

```
var secrand = new SecureRandom();
var b = new byte[20];
// заполнить массив подлинно случайными битами:
secrand.setSeed(b);
```

Если не указать генератору случайных чисел никакого начального значения, он сформирует собственное 20-байтовое значение, запустив несколько потоков на исполнение, переведя их в режим ожидания и рассчитав точное время их выхода из режима ожидания.

НА ЗАМЕТКУ! Трудно сказать, является ли такой алгоритм безопасным. Раньше алгоритмы, действие которых основывалось на вычислении временных показателей некоторых компонентов компьютера, например времени доступа к жесткому диску, зачастую оказывались на поверку генерирующими недостаточно случайные числа.

В примере программы из листинга 10.17 демонстрируется применение алгоритма шифрования AES. Служебный метод шифрования `crypt()` из листинга 10.18 будет еще не раз использован в остальных примерах далее в этой главе. Чтобы воспользоваться данной программой, необходимо сначала сгенерировать секретный ключ, выполнив следующую команду:

```
java aes.AESTest -genkey secret.key
```

Полученный в итоге секретный ключ следует сохранить в файле `secret.key`. После этого шифрование можно выполнить по следующей команде:

```
java aes.AESTest -encrypt plaintextFile \
     encryptedFile secret.key
```

Расшифровывание производится по такой команде:

```
java aes.AESTest -decrypt encryptedFile \
     decryptedFile secret.key
```

Рассматриваемая здесь программа довольно проста. Сначала с помощью параметра `-genkey` генерируется новый секретный ключ, который затем сохраняется в указанном файле. Эта операция обычно занимает много времени, поскольку большая ее часть уходит на инициализацию генератора случайных чисел. Параметры `-encrypt` и `-decrypt` предусматривают вызов одного и того же метода `crypt()`, который, в свою очередь, вызывает такие методы для объекта шифрования, как `update()` и `doFinal()`. Но следует иметь в виду, что метод `update()` вызывается повторно до тех пор, пока блоки вводимых данных не достигнут полной длины, и что метод `doFinal()` вызывается с частично заполненным блоком вводимых данных (который затем заполняется) или вообще безо всяких дополнительных данных (для формирования одного заполняющего блока).

Листинг 10.17. Исходный код из файла **aes/AESTest.java**

```
1  package aes;
2
3  import java.io.*;
4  import java.security.*;
5  import javax.crypto.*;
6
7  /**
8   * В этой программе проверяется шифрование
9   * по алгоритму AES. Применение:
10  * java aes.AESTest -genkey keyfile
```

```
 * java aes.AESTest -encrypt plaintext encrypted keyfile
 * java aes.AESTest -decrypt encrypted decrypted keyfile
 * @author Cay Horstmann
 * @version 1.02 2018-05-01
 */
public class AESTest
{
  public static void main(String[] args)
        throws IOException, GeneralSecurityException,
               ClassNotFoundException
  {
    if (args[0].equals("-genkey"))
    {
      KeyGenerator keygen =
                  KeyGenerator.getInstance("AES");
      var random = new SecureRandom();
      keygen.init(random);
      SecretKey key = keygen.generateKey();
      try (var out = new ObjectOutputStream(
           new FileOutputStream(args[1])))
      {
        out.writeObject(key);
      }
    }
    else
    {
      int mode;
      if (args[0].equals("-encrypt"))
        mode = Cipher.ENCRYPT_MODE;
      else mode = Cipher.DECRYPT_MODE;

      try (var keyIn = new ObjectInputStream(
                       new FileInputStream(args[3]));
           var in = new FileInputStream(args[1]);
           var out = new FileOutputStream(args[2]))
      {
        var key = (Key) keyIn.readObject();
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(mode, key);
        Util.crypt(in, out, cipher);
      }
    }
  }
}
```

Листинг 10.18. Исходный код из файла **aes/Util.java**

```
package aes;

import java.io.*;
import java.security.*;
import javax.crypto.*;

```

```
7  public class Util
8  {
9    /**
10    * Использует шифр для преобразования байтов из
11    * потока ввода и направляет преобразованные байты
12    * в поток вывода
13    * @param in Поток ввода
14    * @param out Поток вывода
15    * @param cipher Шифр для преобразования байтов
16   */
17   public static void crypt(InputStream in,
18         OutputStream out, Cipher cipher)
19         throws IOException, GeneralSecurityException
20   {
21     int blockSize = cipher.getBlockSize();
22     int outputSize = cipher.getOutputSize(blockSize);
23     var inBytes = new byte[blockSize];
24     var outBytes = new byte[outputSize];
25     int inLength = 0;
26
27     var done = false;
28     while (!done)
29     {
30       inLength = in.read(inBytes);
31       if (inLength == blockSize)
32       {
33         int outLength = cipher.update(inBytes, 0,
34                                 blockSize, outBytes);
35         out.write(outBytes, 0, outLength);
36       }
37       else done = true;
38     }
39     if (inLength > 0)
40       outBytes = cipher.doFinal(inBytes, 0, inLength);
41     else
42       outBytes = cipher.doFinal();
43     out.write(outBytes);
44   }
45 }
```

javax.crypto.Cipher 1.4

- **static Cipher getInstance(String *algorithmName*)**
- **static Cipher getInstance(String *algorithmName*, String *providerName*)**

  Возвращают объект типа **Cipher**, реализующий указанный алгоритм шифрования. Если же такой алгоритм не поддерживается, то генерируется исключение типа **NoSuchAlgorithmException**.

javax.crypto.Cipher 1.4 *(окончание)*

- **int getBlockSize()**

  Возвращает размер шифруемого блока в байтах или нулевое значение, если алгоритм шифрования не предусматривает манипулирование блоками данных.

- **int getOutputSize(int *inputLength*)**

  Возвращает размер выходного буфера данных, который требуется, если следующие входные данные имеют количество байтов, определяемое параметром ***inputLength***. В этом методе учитываются любые байты, буферизированные в объекте шифрования.

- **void init(int *mode*, Key *key*)**

  Инициализирует объект, реализующий алгоритм шифрования. Параметр режима может принимать значение одной из следующих констант: **ENCRYPT_MODE**, **DECRYPT_MODE**, **WRAP_MODE** или **UNWRAP_MODE**.

- **byte[] update(byte[] *in*)**
- **byte[] update(byte[] *in*, int *offset*, int *length*)**
- **int update(byte[] *in*, int *offset*, int *length*, byte[] *out*)**

  Преобразуют один блок входных данных. Первые два метода возвращают выходные данные, а третий метод возвращает сведения о количестве байтов, которые были размещены в массиве ***out***.

- **byte[] doFinal()**
- **byte[] doFinal(byte[] *in*)**
- **byte[] doFinal(byte[] *in*, int *offset*, int *length*)**
- **int doFinal(byte[] *in*, int *offset*, int *length*, byte[] *out*)**

  Преобразуют последний блок входных данных и очищают буфер данного объекта шифрования. Первые три метода возвращают выходные данные, а четвертый метод возвращает сведения о количестве байтов, которые были размещены в массиве ***out***.

javax.crypto.KeyGenerator 1.4

- **static KeyGenerator getInstance(String *algorithmName*)**

  Возвращает объект типа **KeyGenerator**, реализующий указанный алгоритм шифрования. Если такой алгоритм не поддерживается, то генерируется исключение типа **NoSuchAlgorithmException**.

- **void init(SecureRandom *random*)**
- **void init(int *keySize*, SecureRandom *random*)**

  Инициализируют генератор ключей шифрования.

- **SecretKey generateKey()**

  Генерирует новый ключ шифрования.

javax.crypto.spec.SecretKeySpec 1.4

- **SecretKeySpec(byte[]*key*, String *algorithmName*)**
  Формирует новый секретный ключ по указанной спецификации.

10.5.3. Потоки шифрования

В библиотеке JCE имеется удобный набор классов, реализующих потоки ввода-вывода и способных автоматически шифровать и расшифровывать данные из этих потоков. В качестве примера ниже показано, как с помощью одного из таких классов зашифровать данные и вывести их в файл.

```
Cipher cipher = . . .;
cipher.init(Cipher.ENCRYPT_MODE, key);
var out = new CipherOutputStream(
          new FileOutputStream(outputFileName), cipher);
var bytes = new byte[BLOCKSIZE];
int inLength = getData(bytes); // получить данные из источника
while (inLength != -1)
{
  out.write(bytes, 0, inLength);
  // получить дополнительные данные из источника:
  inLength = getData(bytes);
}
out.flush();
```

Подобным образом для чтения данных из файла и их расшифровки можно применить класс `CipherInputStream`:

```
Cipher cipher = . . .;
cipher.init(Cipher.DECRYPT_MODE, key);
var in = new CipherInputStream(
         new FileInputStream(inputFileName), cipher);
var bytes = new byte[BLOCKSIZE];
int inLength = in.read(bytes);
while (inLength != -1)
{
  // вывести данные по месту назначения:
  putData(bytes, inLength);
  inLength = in.read(bytes);
}
```

Классы потоков шифрования прозрачно обрабатывают вызовы методов `update()` и `doFinal()`, что, безусловно, очень удобно.

javax.crypto.CipherInputStream 1.4

- **CipherInputStream(InputStream *in*, Cipher *cipher*)**
  Создает поток ввода, который считывает данные из потока ввода ***in*** и расшифровывает или зашифровывает их, используя указанный шифр.

javax.crypto.CipherInputStream 1.4 *(окончание)*

- **int read()**
- **int read(byte[] *b*, int *off*, int *len*)**

  Считывают данные из потока ввода и автоматически расшифровывают или зашифровывают их.

javax.crypto.CipherOutputStream 1.4

- **CipherOutputStream(OutputStream *out*, Cipher *cipher*)**

  Создает поток вывода, направляющий данные в заданный поток вывода ***out***, и зашифровывает или расшифровывает их, используя указанный шифр.
- **void write(int *ch*)**
- **void write(byte[] *b*, int *off*, int *len*)**

  Направляют данные в поток вывода и автоматически зашифровывают или расшифровывают их.
- **void flush()**

  Выводит данные из буфера шифрования, очищая его и заполняя при необходимости недостающие биты.

10.5.4. Шифрование открытым ключом

Алгоритмы шифрования DES и AES, рассматривавшиеся в предыдущем разделе, являются *симметричными*. Это означает, что один и тот же ключ применяется как для шифрования, так и для расшифровывания. Уязвимым местом всех симметричных алгоритмов шифрования является передача ключа. Так, если Алиса отправляет Бобу зашифрованное сообщение, для его расшифровки Бобу требуется тот же самый ключ, которым пользовалась Алиса. При изменении ключа Алиса снова должна отправить Бобу вместе с сообщением новую версию этого ключа по какому-нибудь защищенному каналу. Но если у нее нет доступа к такому каналу связи с Бобом, то ей придется сначала зашифровать все отправляемые ему сообщения.

В качестве выхода из этого положения можно воспользоваться шифрованием открытым ключом. В таком случае у Боба будет пара ключей: открытый и соответствующий ему секретный ключ. Он сможет передавать свой открытый ключ кому угодно, а секретный ключ хранить в тайне. Алисе же останется только использовать этот открытый ключ для шифрования всех своих сообщений Бобу.

Но, к сожалению, все не так просто. Дело в том, что все алгоритмы шифрования открытым ключом действуют *намного* медленнее, чем алгоритмы шифрования симметричными ключами вроде DES или AES. Поэтому было бы неэффективно и непрактично использовать открытые ключи для шифрования данных большого объема. Впрочем, это затруднение нетрудно разрешить,

сочетая шифрование открытым ключом с симметричным шифрованием. Обратимся за разъяснением к следующему примеру.

1. Алиса формирует произвольный симметричный ключ и шифрует им свое сообщение.
2. Используя открытый ключ Боба, Алиса зашифровывает этот симметричный ключ.
3. Алиса посылает Бобу зашифрованное сообщение вместе с зашифрованным симметричным ключом.
4. Используя секретный ключ, Боб расшифровывает симметричный ключ.
5. Используя расшифрованный симметричный ключ, Боб расшифровывает полученное сообщение.

В данном примере никто, кроме Боба, не сможет расшифровать симметричный ключ, потому что секретный ключ имеется только у него. Таким образом, неэффективный алгоритм шифрования открытым ключом применяется только для небольшого объема данных симметричного ключа.

Наиболее распространенным для шифрования открытым ключом является алгоритм RSA, изобретенный Райвестом, Шамиром и Адлеманом. До октября 2000 года этот алгоритм был защищен патентом, выданным компании RSA Security Inc., а для получения лицензии на его использование приходилось платить 3%-ную пошлину с минимальным ежегодным взносом 50 тыс. долларов США. В настоящее время этот алгоритм уже не является коммерческим и стал всеобщим достоянием.

Чтобы воспользоваться алгоритмом шифрования RSA, необходимо создать открытый и секретный ключи средствами класса `KeyPairGenerator`:

```
KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
var random = new SecureRandom();
pairgen.initialize(KEYSIZE, random);
KeyPair keyPair = pairgen.generateKeyPair();
Key publicKey = keyPair.getPublic();
Key privateKey = keyPair.getPrivate();
```

Для запуска на выполнение примера программы из листинга 10.19 применяются три параметра. В частности, параметр `-genkey` служит для создания пары ключей, параметр `-encrypt` — для генерирования ключа по алгоритму шифрования AES и его свертывания с помощью открытого ключа, как показано ниже.

```
Key key = . . .; // ключ по алгоритму шифрования AES
Key publicKey = . . .; // открытый ключ по алгоритму
                       // шифрования RSA
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.WRAP_MODE, publicKey);
byte[] wrappedKey = cipher.wrap(key);
```

Далее создается файл, состоящий из следующих элементов.

- Длина свернутого ключа.
- Байты свернутого ключа.
- Открытый текст, зашифрованный ключом по алгоритму AES.

Третий параметр, `-decrypt`, служит для расшифровывания файла. Чтобы опробовать данную программу, необходимо сначала создать ключи по алгоритму шифрования RSA, выполнив следующую команду:

```
java rsa.RSATest -genkey public.key private.key
```

Затем следует зашифровать файл по команде:

```
java rsa.RSATest -encrypt plaintextFile \
    encryptedFile public.key
```

Наконец, файл следует расшифровать и проверить, совпадают ли полученные данные с ранее зашифрованными, введя команду

```
java rsa.RSATest -decrypt encryptedFile \
    decryptedFile private.key
```

Листинг 10.19. Исходный код из файла **rsa/RSATest.java**

```
1  package rsa;
2
3  import java.io.*;
4  import java.security.*;
5  import javax.crypto.*;
6
7  /**
8   * В этой программе проверяется шифрование
9   * по алгоритму RSA. Применение:
10  * java rsa.RSATest -genkey public private
11  * java rsa.RSATest -encrypt plaintext encrypted public
12  * java rsa.RSATest -decrypt encrypted decrypted private
13  * @author Cay Horstmann
14  * @version 1.02 2018-05-01
15  */
16 public class RSATest
17 {
18   private static final int KEYSIZE = 512;
19
21   public static void main(String[] args)
22         throws IOException, GeneralSecurityException,
23                ClassNotFoundException
24   {
25     if (args[0].equals("-genkey"))
26     {
27       KeyPairGenerator pairgen =
28             KeyPairGenerator.getInstance("RSA");
29       var random = new SecureRandom();
30       pairgen.initialize(KEYSIZE, random);
31       KeyPair keyPair = pairgen.generateKeyPair();
32       try (var out = new ObjectOutputStream(
```

```
33            new FileOutputStream(args[1])))
34       {
35         out.writeObject(keyPair.getPublic());
36       }
37       try (var out = new ObjectOutputStream(
38                new FileOutputStream(args[2])))
39       {
40         out.writeObject(keyPair.getPrivate());
41       }
42     }
43     else if (args[0].equals("-encrypt"))
44     {
45       KeyGenerator keygen =
46                   KeyGenerator.getInstance("AES");
47       var random = new SecureRandom();
48       keygen.init(random);
49       SecretKey key = keygen.generateKey();
50
51       // свернуть с помощью открытого ключа
52       // по алгоритму шифрования RSA:
53       try (var keyIn = new ObjectInputStream(
54                        new FileInputStream(args[3]));
55            var out = new DataOutputStream(
56                      new FileOutputStream(args[2]));
57            var in = new FileInputStream(args[1]))
58       {
59         var publicKey = (Key) keyIn.readObject();
60         Cipher cipher = Cipher.getInstance("RSA");
61         cipher.init(Cipher.WRAP_MODE, publicKey);
62         byte[] wrappedKey = cipher.wrap(key);
63         out.writeInt(wrappedKey.length);
64         out.write(wrappedKey);
65
66         cipher = Cipher.getInstance("AES");
67         cipher.init(Cipher.ENCRYPT_MODE, key);
68         Util.crypt(in, out, cipher);
69       }
70     }
71     else
72     {
73       try (var in = new DataInputStream(
74                     new FileInputStream(args[1]));
75            var keyIn = new ObjectInputStream(
76                        new FileInputStream(args[3]));
77            var out = new FileOutputStream(args[2]))
78       {
79         int length = in.readInt();
80         var wrappedKey = new byte[length];
81         in.read(wrappedKey, 0, length);
82
81         // развернуть с помощью секретного ключа
82         // по алгоритму шифрования RSA
83         var privateKey = (Key) keyIn.readObject();
84
85         Cipher cipher = Cipher.getInstance("RSA");
86         cipher.init(Cipher.UNWRAP_MODE, privateKey);
```

```
87          Key key = cipher.unwrap(wrappedKey, "AES",
88                                  Cipher.SECRET_KEY);
89
90          cipher = Cipher.getInstance("AES");
91          cipher.init(Cipher.DECRYPT_MODE, key);
92
93          Util.crypt(in, out, cipher);
94        }
95      }
96    }
97 }
```

В этой главе было показано, каким образом модель безопасности обеспечивает контролируемое выполнение кода, что является отличительной и очень важной особенностью платформы Java. Кроме того, были рассмотрены различные службы, предоставляемые в библиотеке Java для аутентификации и шифрования.

В следующей главе будут рассмотрены расширенные возможности для программирования средствами библиотеки Swing.

ГЛАВА 11

Расширенные средства Swing и графика

В этой главе...

- Таблицы
- Деревья
- Расширенные средства AWT
- Растровые изображения
- Вывод изображений на печать

В этой главе продолжается начатое в первом томе настоящего издания обсуждение набора компонентов из библиотек Swing и AWT, предназначенных для построения графического пользовательского интерфейса и графики. Основное внимание в ней уделяется разработке элементов пользовательского интерфейса на стороне клиента, а также формированию изображений и графики на стороне сервера. В состав библиотеки Swing входят сложные, логически развитые компоненты для воспроизведения таблиц и деревьев. А с помощью прикладного интерфейса API для графики можно воспроизводить векторную графику практически любой сложности. Наконец, применяя прикладной интерфейс API для вывода на печать, можно формировать отпечатки и файлы формата PostScript.

11.1. Таблицы

Компонент `JTable` служит для отображения таблицы в виде двухмерной сетки объектов. Таблицы широко применяются при построении графического пользовательского интерфейса, поэтому разработчики библиотеки Swing

уделили должное внимание созданию данного компонента. Компонент `JTable` имеет очень сложную структуру, но она, в отличие от других компонентов Swing, практически скрыта от пользователей. С помощью всего нескольких строк кода можно создать полноценную таблицу. Но для составления специальных таблиц придется написать дополнительный код, оформить их особый внешний вид и задать конкретное их поведение в разрабатываемой прикладной программе.

В этом разделе поясняется, каким образом создаются простые таблицы, как организуется взаимодействие пользователей с ними и как они чаще всего настраиваются. Аналогично другим сложным компонентам Swing, все особенности манипулирования таблицами невозможно подробно описать в одном разделе. Поэтому для углубленного изучения данного вопроса рекомендуется следующая дополнительная литература: книга *Graphic Java™: Mastering the JFC, Volume II: Swing, 3rd Edition* Дэвида М. Гери (David M. Geary; издательство Prentice Hall, 1999 г.) или книга *Core Swing* Кима Топли (Kim Topley; издательство Prentice Hall, 1999 г.).

11.1.1. Простая таблица

Компонент `JTable` не хранит свои данные, а получает их из *модели таблицы*. Класс `JTable` содержит конструктор, который заключает двухмерный массив объектов в оболочку модели, используемой по умолчанию. Именно такой способ применяется в рассматриваемом здесь первом примере создания таблиц. Модели таблиц более подробно обсуждаются далее в этой главе.

На рис. 11.1 приведена типичная таблица с описаниями свойств планет Солнечной системы. Следует иметь в виду, что в столбце `Gaseous` указывается логическое значение `true`, если планета состоит в основном из водорода и гелия, а иначе — логическое значение `false`. Значения в столбце `Color` пока еще не имеют какого-то определенного смысла, но этот столбец понадобится в следующих примерах.

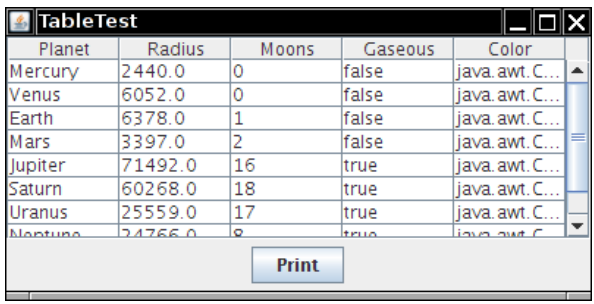


Рис. 11.1. Пример простой таблицы

Как следует из приведенного ниже фрагмента кода, взятого из листинга 11.1, в котором представлен исходный код рассматриваемого здесь примера программы, данные таблицы хранятся в виде двухмерного массива значений типа `Object`.

```
Object[][] cells =
{
  { "Mercury", 2440.0, 0, false, Color.YELLOW },
  { "Venus", 6052.0, 0, false, Color.YELLOW },
  . . .
}
```

НА ЗАМЕТКУ! В данном случае используются преимущества автоупаковки. Так, значения примитивных типов во втором, третьем и четвертом столбцах автоматически преобразуются в объекты типа **Double**, **Integer** и **Boolean**.

Для отображения каждого объекта в таблице вызывается метод `toString()`. Поэтому цвета в последнем столбце представлены в виде строк `java.awt.Color[r=...,g=...,b=...]`.

Сначала имена столбцов задаются в отдельном массиве символьных строк следующим образом:

```
String[] columnNames = { "Planet", "Radius", "Moons",
                         "Gaseous", "Color" };
```

Затем из массивов ячеек и имен столбцов составляется таблица:

```
var table = new JTable(cells, columnNames);
```

Таблицу можно также снабдить полосами прокрутки. Для этого достаточно заключить ее в оболочку компонента `JScrollPane`, как показано ниже. При прокрутке таблицы ее заголовок не исчезает из виду.

```
var pane = new JScrollPane(table);
```

Если щелкнуть кнопкой мыши на одном из столбцов и перетащить его влево или вправо, весь столбец визуально отделяется от других столбцов небольшим промежутком (рис. 11.2). Перетаскиваемый столбец можно опустить на любом другом месте. Такое переупорядочение столбцов возможно *только* в представлении таблицы, и оно никак не влияет на модель данных.

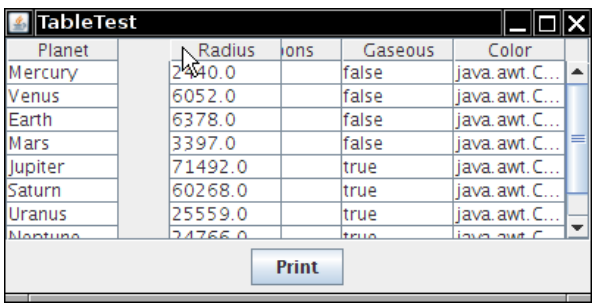


Рис. 11.2. Перемещение столбца в таблице

Чтобы изменить размеры столбца, достаточно навести курсор на границу раздела двух столбцов. Как только указатель мыши примет вид двойной стрелки, останется лишь перетащить границу раздела столбцов в нужное место (рис. 11.3).

Чтобы выбрать строку в таблице, достаточно щелкнуть на любом месте в этой строке. Выбранные строки таблицы выделяются другим цветом. Далее будет показано, каким образом организуется обработка событий выбора из таблицы. В рассматриваемом здесь примере допускается редактирование ячеек таблицы, но внесенные в них изменения не сохраняются. Поэтому при разработке собственной прикладной программы вам придется выбирать одно из двух: вообще запретить редактирование в ячейках таблицы или организовать обработку

событий редактирования и обновление модели таблицы. Подробнее об этом — далее в настоящем разделе.

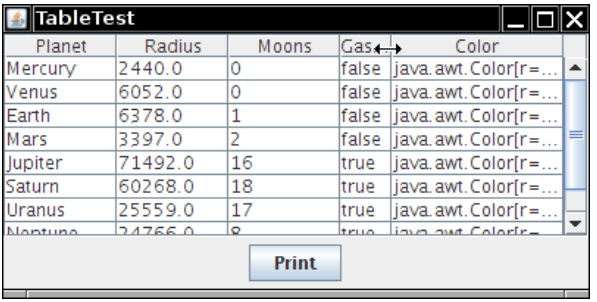


Рис. 11.3. Изменение размеров столбцов в таблице

Наконец, если щелкнуть на заголовке столбца, строки таблицы будут автоматически отсортированы. Если щелкнуть еще раз, сортировка будет произведена в обратном порядке. Такое поведение активизируется при вызове следующего метода:

```
table.setAutoCreateRowSorter(true);
```

Для вывода таблицы на печать вызывается следующий метод:

```
table.print();
```

ВНИМАНИЕ! Если таблица не вписывается в панель прокрутки, ее заголовок придется добавить явным образом, сделав следующий вызов:

```
add(table.getTableHeader(), BorderLayout.NORTH);
```

Листинг 11.1. Исходный код из файла **table/TableTest.java**

```
1  package table;
2
3  import java.awt.*;
4  import java.awt.print.*;
5
6  import javax.swing.*;
7
8  /**
9   * В этой программе демонстрируется порядок
10  * отображения простой таблицы
11  * @version 1.14 2018-05-01
12  * @author Cay Horstmann
13  */
14 public class TableTest
15 {
16   public static void main(String[] args)
17   {
18     EventQueue.invokeLater(() ->
19       {
20         var frame = new PlanetTableFrame();
21         frame.setTitle("TableTest");
```

```
22         frame.setDefaultCloseOperation(
23               JFrame.EXIT_ON_CLOSE);
24         frame.setVisible(true);
25       });
26    }
27 }
28
29 /**
30  * Этот фрейм содержит таблицу с данными о планетах
31  */
32 class PlanetTableFrame extends JFrame
33 {
34    private String[] columnNames = { "Planet", "Radius",
35                   "Moons", "Gaseous", "Color" };
36    private Object[][] cells =
37    {
38      { "Mercury", 2440.0, 0, false, Color.YELLOW },
39      { "Venus", 6052.0, 0, false, Color.YELLOW },
40      { "Earth", 6378.0, 1, false, Color.BLUE },
41      { "Mars", 3397.0, 2, false, Color.RED },
42      { "Jupiter", 71492.0, 16, true, Color.ORANGE },
43      { "Saturn", 60268.0, 18, true, Color.ORANGE },
44      { "Uranus", 25559.0, 17, true, Color.BLUE },
45      { "Neptune", 24766.0, 8, true, Color.BLUE },
46      { "Pluto", 1137.0, 1, false, Color.BLACK }
47    };
48
49    public PlanetTableFrame()
50    {
51      var table = new JTable(cells, columnNames);
52      table.setAutoCreateRowSorter(true);
53      add(new JScrollPane(table), BorderLayout.CENTER);
54      var printButton = new JButton("Print");
55      printButton.addActionListener(event ->
56        {
57          try { table.print(); }
58          catch (SecurityException | PrinterException ex)
59          { ex.printStackTrace(); }
60        });
61      var buttonPanel = new JPanel();
62      buttonPanel.add(printButton);
63      add(buttonPanel, BorderLayout.SOUTH);
64      pack();
65    }
66 }
```

javax.swing.JTable 1.2

- **JTable(Object[][] *entries*, Object[] *columnNames*)**
  Составляет таблицу с моделью, используемой по умолчанию.
- **void print()** 5.0
  Отображает диалоговое окно печати и выводит таблицу на печать.

javax.swing.JTable 1.2 *(окончание)*

- **boolean getAutoCreateRowSorter()** 6
- **void setAutoCreateRowSorter(boolean *newValue*)** 6

  Получают или устанавливают свойство **autoCreateRowSorter**. По умолчанию оно принимает логическое значение **false**. Если установлено именно это логическое значение, то при любом изменении модели таблицы будет автоматически устанавливаться сортировщик строк, выбираемый по умолчанию.
- **boolean getFillsViewportHeight()** 6
- **void setFillsViewportHeight(boolean *newValue*)** 6

  Получают или устанавливают свойство **fillsViewportHeight**. По умолчанию оно принимает логическое значение **false**. Если установлено именно это логическое значение, то таблица всегда будет заполнять объемлющую область просмотра.

11.1.2. Модели таблиц

Описанные в предыдущем разделе данные таблицы хранились в виде двухмерного массива. Но такой способ в прикладном коде обычно не применяется. Вместо вывода данных в массив для их отображения в табличном виде рекомендуется реализовать собственную модель таблицы.

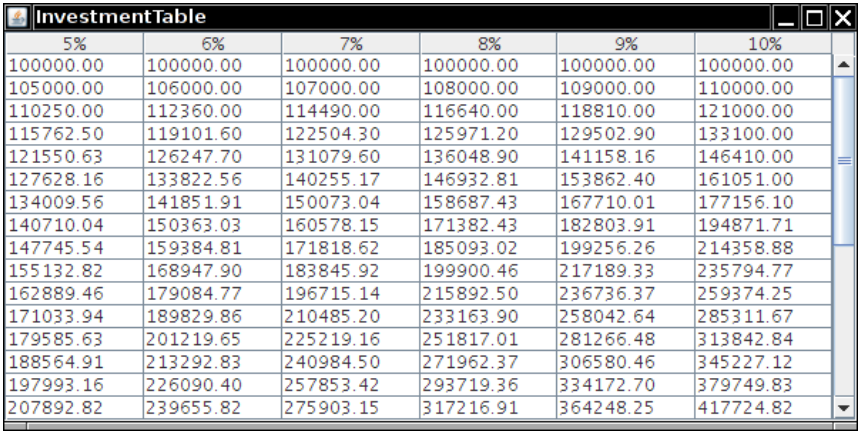
Реализовать модель таблицы совсем не трудно, поскольку для этой цели предусмотрен отдельный класс `AbstractTableModel` с большинством требующихся методов. Остается лишь предоставить следующие три метода:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

Метод `getValueAt()` можно реализовать несколькими способами. Так, если требуется отобразить набор строк типа `RowSet` из таблицы, содержащий результат запроса к базе данных, то подойдет следующий вариант реализации данного метода:

```
public Object getValueAt(int r, int c)
{
  try
  {
    rowSet.absolute(r + 1);
    return rowSet.getObject(c + 1);
  }
  catch (SQLException e)
  {
    e.printStackTrace();
    return null;
  }
}
```

Следующий пример программы еще проще. В ней составляется таблица из рассчитанных значений, а именно: роста инвестиций при разных учетных ставках (рис. 11.4).



5%	6%	7%	8%	9%	10%
100000.00	100000.00	100000.00	100000.00	100000.00	100000.00
105000.00	106000.00	107000.00	108000.00	109000.00	110000.00
110250.00	112360.00	114490.00	116640.00	118810.00	121000.00
115762.50	119101.60	122504.30	125971.20	129502.90	133100.00
121550.63	126247.70	131079.60	136048.90	141158.16	146410.00
127628.16	133822.56	140255.17	146932.81	153862.40	161051.00
134009.56	141851.91	150073.04	158687.43	167710.01	177156.10
140710.04	150363.03	160578.15	171382.43	182803.91	194871.71
147745.54	159384.81	171818.62	185093.02	199256.26	214358.88
155132.82	168947.90	183845.92	199900.46	217189.33	235794.77
162889.46	179084.77	196715.14	215892.50	236736.37	259374.25
171033.94	189829.86	210485.20	233163.90	258042.64	285311.67
179585.63	201219.65	225219.16	251817.01	281266.48	313842.84
188564.91	213292.83	240984.50	271962.37	306580.46	345227.12
197993.16	226090.40	257853.42	293719.36	334172.70	379749.83
207892.82	239655.82	275903.15	317216.91	364248.25	417724.82

Рис. 11.4. Пример таблицы с данными о росте инвестиций при разных учетных ставках

В данном примере метод `getValueAt()` выполняет расчет соответствующего значения и форматирует его, как показано ниже.

```
public Object getValueAt(int r, int c)
{
  double rate = (c + minRate) / 100.0;
  int nperiods = r;
  double futureBalance =
         INITIAL_BALANCE * Math.pow(1 + rate, nperiods);
  return String.format("%.2f", futureBalance);
}
```

Методы `getRowCount()` и `getColumnCount()` просто возвращают количество строк и столбцов соответственно:

```
public int getRowCount() { return years; }
public int getColumnCount()
{ return maxRate - minRate + 1; }
```

Если заголовки столбцов не заданы явно, то для них в методе `getColumnName()` из класса `AbstractTableModel` по умолчанию используются имена `A`, `B`, `C` и т.д. Для изменения этих имен следует переопределить метод `getColumnName()`. В данном примере в качестве заголовка столбца используется учетная ставка, как показано ниже. Весь исходный код программы из данного примера представлен в листинге 11.2.

```
public String getColumnName(int c)
{ return (c + minRate) + "%"; }
```

Листинг 11.2. Исходный код из файла **`tableModel/InvestmentTable.java`**

```
1  package tableModel;
2
3  import java.awt.*;
4
5  import javax.swing.*;
```

```
6  import javax.swing.table.*;
7
8  /**
9   * В этой программе демонстрируется построение
10  * таблицы по ее модели
11  * @version 1.04 2018-05-01
12  * @author Cay Horstmann
13  */
14 public class InvestmentTable
15 {
16   public static void main(String[] args)
17   {
18     EventQueue.invokeLater(() ->
19       {
20         var frame = new InvestmentTableFrame();
21         frame.setTitle("InvestmentTable");
22         frame.setDefaultCloseOperation(
23               JFrame.EXIT_ON_CLOSE);
24         frame.setVisible(true);
25       });
26   }
27 }
28
29 /**
30  * Этот фрейм содержит таблицу капиталовложений
31  */
32 class InvestmentTableFrame extends JFrame
33 {
34   public InvestmentTableFrame()
35   {
36     var model = new InvestmentTableModel(30, 5, 10);
37     var table = new JTable(model);
38     add(new JScrollPane(table));
39     pack();
40   }
41 }
42
43 /**
44   * В этой модели таблицы рассчитывается содержимое
45   * ячеек таблицы всякий раз, когда оно запрашивается.
46   * В таблице представлен рост капиталовложений в
47   * течение ряда лет при разных учетных ставках
48   */
49 class InvestmentTableModel extends AbstractTableModel
50 {
51   private static double INITIAL_BALANCE = 100000.0;
52
53   private int years;
54   private int minRate;
55   private int maxRate;
56
57   /**
58    * Конструирует модель таблицы капиталовложений
59    * @param y Количество лет
60    * @param r1 Наинизшая учетная ставка для
61    *           составления таблицы
```

```
62     * @param r2 Наивысшая учетная ставка для
63     *           составления таблицы
64     */
65    public InvestmentTableModel(int y, int r1, int r2)
66    {
67      years = y;
68      minRate = r1;
69      maxRate = r2;
70    }
71
72    public int getRowCount()
73    {
74      return years;
75    }
76
77    public int getColumnCount()
78    {
79      return maxRate - minRate + 1;
80    }
81
82    public Object getValueAt(int r, int c)
83    {
84      double rate = (c + minRate) / 100.0;
85      int nperiods = r;
86      double futureBalance = INITIAL_BALANCE
87                           * Math.pow(1 + rate, nperiods);
88      return String.format("%.2f", futureBalance);
89    }
90
91    public String getColumnName(int c)
92    {
93      return (c + minRate) + "%";
94    }
95 }
```

javax.swing.table.TableModel 1.2

- **int getRowCount()**
- **int getColumnCount()**

  Получают количество строк и столбцов в модели таблицы.
- **Object getValueAt(int *row*, int *column*)**

  Получает значение из указанных строки и столбца таблицы.
- **void setValueAt(Object *newValue*, int *row*, int *column*)**

  Устанавливает новое значение в указанных строке и столбце таблицы.
- **boolean isCellEditable(int *row*, int *column*)**

  Возвращает логическое значение **true**, если ячейка в указанных строке и столбце таблицы доступна для редактирования.
- **String getColumnName(int *column*)**

  Получает заголовок столбца таблицы.

11.1.3. Манипулирование строками и столбцами таблицы

В этом разделе рассматриваются способы манипулирования строками и столбцами таблицы. Таблица, составленная средствами Swing, имеет несимметричную структуру, допускающую выполнение разных операций над строками и столбцами. Дело в том, что компонент `JTable`, реализующий таблицу в библиотеке Swing, предназначен для отображения строк одинаковой структуры, например, данных из таблиц базы данных, а не объектов в виде произвольной двухмерной сетки. Такая асимметрия демонстрируется в данном разделе на конкретных примерах.

11.1.3.1. Классы столбцов таблицы

В следующем примере снова рассматривается таблица с данными о планетах, но теперь особое внимание уделяется типам данных в столбцах. В частности, приведенный ниже метод из модели таблицы возвращает класс с описанием типа столбца. Эти сведения используются в классе `JTable` для выбора средства воспроизведения, подходящего для конкретного класса столбца.

```
Class<?> getColumnClass(int columnIndex)
```

В табл. 11.1 перечислены действия, выполняемые по умолчанию, для воспроизведения столбцов по их типам (и соответствующим классам). Так, в столбцах таблицы на рис. 11.5 показаны флажки и изображения. Автор выражает искреннюю благодарность Джиму Эвинсу (Jim Evins) за любезно представленные изображения планет. Чтобы воспроизвести в таблице столбцы других типов, следует установить специальное средство воспроизведения, как поясняется далее, в разделе 11.1.4.

Таблица 11.1. Действия, выполняемые по умолчанию, для воспроизведения столбцов таблицы по их типам

Тип	Как воспроизводится
`Boolean`	Флажок
`Icon`	Изображение
`Object`	Символьная строка

11.1.3.2. Доступ к столбцам таблицы

Компонент `JTable` сохраняет сведения обо всех столбцах таблицы в объектах типа `TableColumn`, а объект типа `TableColumnModel` манипулирует столбцами. (На рис. 11.6 схематически показаны отношения между наиболее важными классами таблиц.) Если столбцы таблицы не предполагается перемещать или вставлять динамически, то обращаться к модели столбцов таблицы (и соответствующему классу) придется нечасто. Чаще всего обращаться к этой модели требуется для получения объекта типа `TableColumn`, как показано ниже.

```
int columnIndex = . . .;
TableColumn column = table.getColumnModel()
                          .getColumn(columnIndex);
```

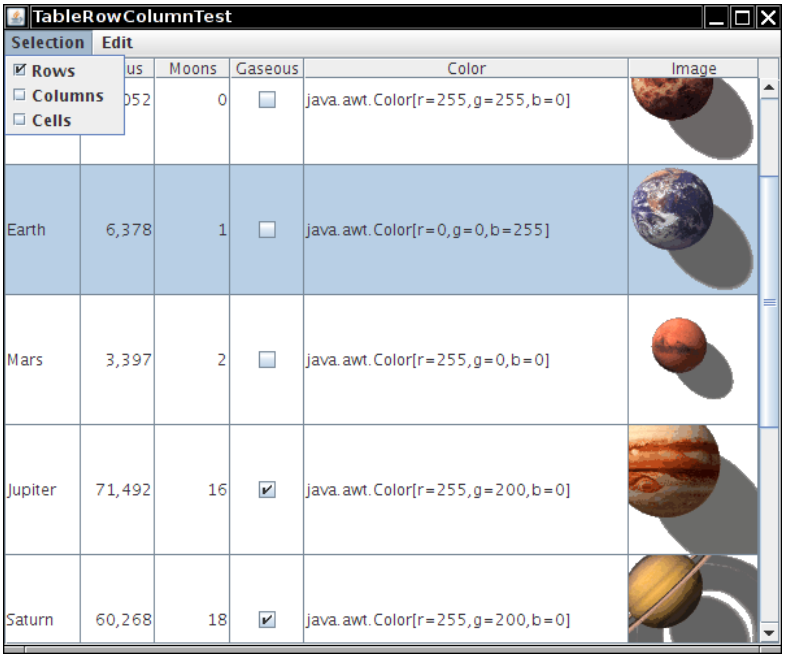


Рис. 11.5. Пример таблицы с данными о планетах, в столбцах которой воспроизводятся флажки и изображения планет

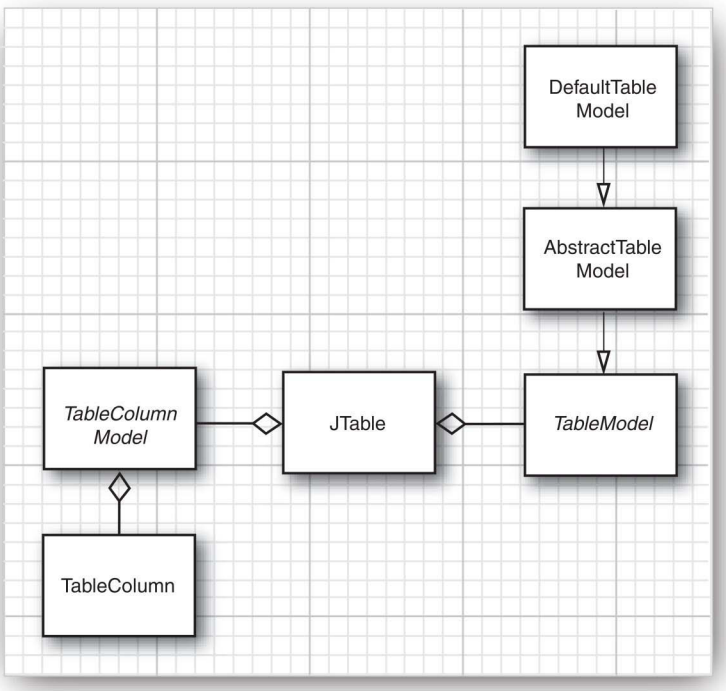


Рис. 11.6. Отношения между табличными классами

11.1.3.3. Изменение размеров столбцов таблицы

Класс `TableColumn` позволяет изменять размеры столбцов. В частности, для указания предпочтительной, минимальной и максимальной ширины столбца используются следующие методы:

```
void setPreferredWidth(int width)
void setMinWidth(int width)
void setMaxWidth(int width)
```

Полученная в итоге информация используется компонентом составления таблиц для размещения столбцов. Чтобы разрешить или запретить пользователю изменять размеры столбцов, достаточно вызвать метод `void setResizable(boolean resizable)`, а для того чтобы изменить размеры столбца программным путем — метод `void setWidth(int width)`.

Если изменяются размеры одного столбца, то по умолчанию общие размеры таблицы остаются прежними. Это, конечно, приводит к увеличению или уменьшению размеров остальных столбцов. По умолчанию подобные изменения распространяются на все столбцы, расположенные справа от того столбца, размеры которого изменяются. Благодаря этому требующиеся размеры всех столбцов можно указывать слева направо.

В табл. 11.2 перечислены режимы изменения размеров таблицы при изменении размеров столбца. Их можно указать, вызвав метод `void setAutoResizeMode(int mode)` из класса `JTable`.

Таблица 11.2. Режимы изменения размеров таблицы

Режим	Поведение
AUTO_RESIZE_OFF	Размеры остальных столбцов остаются прежними, а размеры таблицы изменяются
AUTO_RESIZE_NEXT_COLUMN	Изменяются только размеры следующего столбца
AUTO_RESIZE_SUBSEQUENT_COLUMNS	Равномерно изменяются размеры всех последующих столбцов (выбирается по умолчанию)
AUTO_RESIZE_LAST_COLUMN	Изменяются размеры только последнего столбца
AUTO_RESIZE_ALL_COLUMNS	Размеры всех столбцов таблицы изменяются равномерно (это не самый лучший вариант, так как он не дает пользователю возможность подгонять многие столбцы под требующиеся размеры)

11.1.3.4. Изменение размеров строк таблицы

Изменение высоты строк таблицы производится непосредственно в классе `JTable`. Если высота ячеек выше заданной по умолчанию, то ее придется указать явно, как показано ниже.

```
table.setRowHeight(height);
```

По умолчанию все строки таблицы имеют одинаковую высоту, но данную установку можно изменить с помощью метода `setRowHeight()` следующим образом:

```
table.setRowHeight(row, height);
```

Конкретная высота строки задается упомянутыми выше методами за вычетом величины междустрочного интервала. По умолчанию величина этого интервала равна `1`, но любое другое его значение можно указать следующим образом:

```
table.setRowMargin(margin);
```

11.1.3.5. Выбор строк, столбцов и ячеек таблицы

В зависимости от заданного режима пользователь может выбирать строки, столбцы и отдельные ячейки таблицы. По умолчанию допускается выбор строки, т.е. после щелчка кнопкой мыши на одной из ячеек будет выбрана вся строка (см. рис. 11.5). Для отмены режима выбора строк достаточно вызвать следующий метод:

```
table.setRowSelectionAllowed(false)
```

В режиме выбора строк пользователю можно разрешить выбор только одной строки, нескольких смежных или несмежных строк таблицы. Для этого необходимо получить *модель выбора* и вызвать ее метод `setSelectionMode()` следующим образом:

```
table.getSelectionModel().setSelectionMode(mode);
```

В качестве параметра `mode` можно указать следующие значения:

```
ListSelectionModel.SINGLE_SELECTION
ListSelectionModel.SINGLE_INTERVAL_SELECTION
ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
```

По умолчанию режим выбора столбцов отключен. Его можно включить, вызвав следующий метод:

```
table.setColumnSelectionAllowed(true)
```

Одновременное включение режимов выбора строк и столбцов равнозначно включению режима выбора ячеек (рис. 11.7). Эти режимы можно указать явно, вызвав следующий метод:

```
table.setCellSelectionEnabled(true)
```

Чтобы посмотреть, каким образом выбор ячеек осуществляется на практике, запустите на выполнение пример программы, исходный код которой представлен в листинге 11.3. Активизируйте режим выбора строки, столбца или ячейки из меню Selection (Выбор) и понаблюдайте за тем, как изменяется поведение таблицы в данном режиме.

С помощью методов `getSelectedRows()` и `getSelectedColumns()` можно выяснить, какие именно строки и столбцы были выбраны. Оба метода возвращают массив `int[]` индексов выбранных элементов. Следует, однако, иметь в виду, что значения индексов берутся из представления таблицы, а не из базовой модели таблицы. Попытайтесь выбрать строки и столбцы, а затем перетащите столбцы

в разные места и отсортируйте строки, щелкая на заголовках столбцов. Выбрав из меню пункт Print Selection (Выбор для печати), посмотрите, какие строки и столбцы отображаются как выбранные. Если же требуется преобразовать значения индексов таблицы в значения индексов модели таблицы, то воспользуйтесь методами `convertRowIndexToModel()` и `convertColumnIndexToModel()` из класса `JTable`.

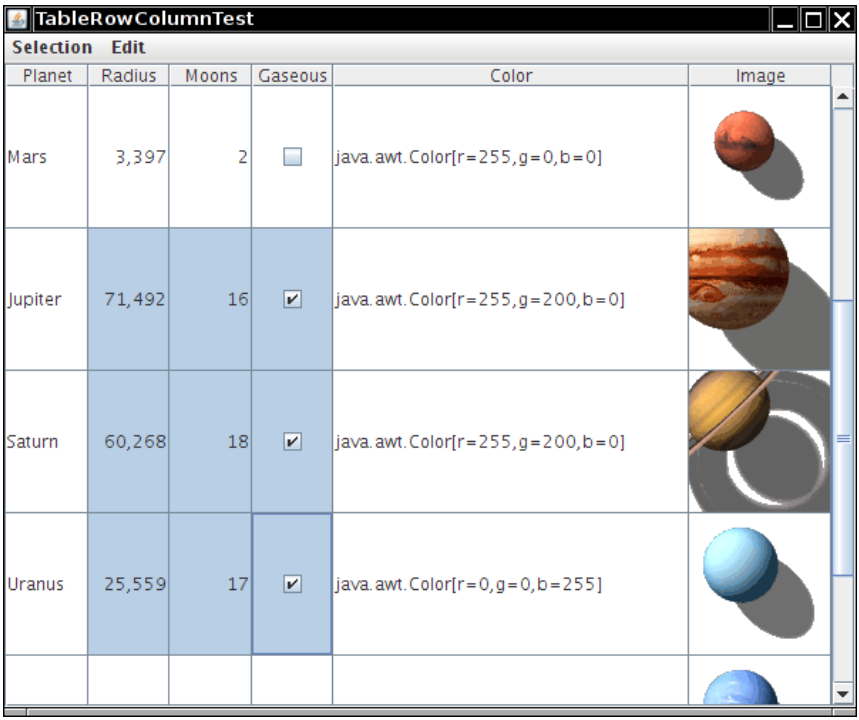


Рис. 11.7. Выбор ряда ячеек в таблице

11.1.3.6. Сортировка строк таблицы

Как было показано в рассмотренном выше первом примере таблицы, компонент `JTable` можно без особого труда дополнить функцией сортировки строк таблицы, вызвав метод `setAutoCreateRowSorter()`. Но для того чтобы получить больше возможностей управлять сортировкой строк таблицы, в компоненте `JTable` следует установить и настроить объект типа `TableRowSorter<M>`. Параметр типа `M` обозначает тип модели; им должен быть подтип интерфейса `TableModel`, как показано ниже.

```
var sorter = new TableRowSorter<TableModel>(model);
table.setRowSorter(sorter);
```

Некоторые столбцы должны быть исключены из сортировки (например, столбец с изображениями в таблице с данными о планетах). Чтобы исключить отдельные столбцы таблицы из сортировки, достаточно вызвать следующий метод:

```
sorter.setSortable(IMAGE_COLUMN, false);
```

Для каждого столбца можно также установить специальное средство сравнения. В рассматриваемом здесь примере предполагается отсортировать цвета в столбце `Color`, отдавая предпочтение синему и зеленому цвету над красным. Если щелкнуть на столбце `Color`, планеты синего цвета окажутся внизу таблицы. Такой результат сортировки по цвету достигается с помощью следующего метода:

```
sorter.setComparator(COLOR_COLUMN, new Comparator<Color>()
  {
    public int compare(Color c1, Color c2)
    {
      int d = c1.getBlue() - c2.getBlue();
      if (d != 0) return d;
      d = c1.getGreen() - c2.getGreen();
      if (d != 0) return d;
      return c1.getRed() - c2.getRed();
    }
})
```

Если не указать компаратор для столбцов, сортировка будет произведена в следующем порядке.

1. Если класс столбца относится к типу `String`, то по умолчанию используется средство сортировки, возвращаемое из метода `Collator.getInstance()`. Это средство сортирует символьные строки в соответствии с текущими региональными настройками. (Подробнее о региональных настройках и средствах сортировки см. в главе 7.)
2. Если класс столбца реализует интерфейс `Comparable`, то используется метод `compareTo()`.
3. Если для сортировки установлен преобразователь типа `TableStringConverter`, то сортировку символьных строк, возвращаемых методом `toString()` этого преобразователя, необходимо выполнять с помощью средства сортировки, выбираемого по умолчанию. Чтобы воспользоваться именно таким способом сортировки, необходимо определить преобразователь символьных строк следующим образом:

```
sorter.setStringConverter(new TableStringConverter()
{
  public String toString(TableModel model,
                         int row, int column)
  {
    Object value = model.getValueAt(row, column);
    преобразовать объект value в символьную строку
    и возвратить ее
  }
});
```

4. В противном случае вызывается метод `toString()` со значениями в ячейках, которые упорядочиваются выбираемым по умолчанию средством сортировки.

11.1.3.7. Фильтрация строк таблицы

Помимо сортировки строк таблицы, класс `TableRowSorter` позволяет избирательно скрывать их. Этот процесс называется *фильтрацией*. Чтобы активизировать режим фильтрации, следует установить соответствующий фильтр типа `RowFilter`. Например, для того чтобы отобрать все строки таблицы, содержащие хотя бы один спутник планеты, достаточно вызвать метод

```
sorter.setRowFilter(RowFilter.numberFilter(
      ComparisonType.NOT_EQUAL, 0, MOONS_COLUMN));
```

Здесь используется предопределенный фильтр чисел. Чтобы создать такой фильтр, понадобятся следующие средства.

- Порядок сравнения (одна из констант `EQUAL`, `NOT_EQUAL`, `AFTER` или `BEFORE`).
- Объект подкласса, производного от класса `Number` (например, `Integer` или `Double`). Допускаются только те объекты, которые имеют тот же класс, что и у данного объекта типа `Number`.
- От нуля и больше значений индекса столбца. Если эти значения не заданы, поиск будет производиться по всем столбцам.

Аналогичным образом в статическом методе `RowFilter.dateFilter()` создается фильтр дат. Отличие состоит лишь в том, что вместо объекта типа `Number` задается объект типа `Date`. Наконец, в статическом методе `RowFilter.regexFilter()` создается фильтр, осуществляющий поиск символьных строк, совпадающих с регулярным выражением. Например, в следующей строке кода отбираются только те планеты, название которых не оканчивается на `"s"`. (Подробнее о регулярных выражениях см. в главе 2.)

```
sorter.setRowFilter(RowFilter.regexFilter(
                    ".*[^s]$", PLANET_COLUMN));
```

Кроме того, фильтры можно применять в различных сочетаниях с помощью методов `andFilter()`, `orFilter()` и `notFilter()`. Так, если требуется отобрать планеты, названия которых не оканчиваются на `"s"` и которые имеют как минимум один спутник, для этого можно применить фильтры в следующем сочетании:

```
sorter.setRowFilter(RowFilter.andFilter(List.of(
      RowFilter.regexFilter(".*[^s]$", PLANET_COLUMN),
      RowFilter.numberFilter(ComparisonType.NOT_EQUAL,
                             0, MOONS_COLUMN))));
```

Чтобы реализовать собственный фильтр, необходимо предоставить объект подкласса, производного от класса `RowFilter`, и реализовать метод `include()` с целью указать те строки таблицы, которые требуется отобрать и отобразить. Сделать это нетрудно, хотя и не так просто в силу обобщенного характера класса `RowFilter`.

У класса `RowFilter<M, I>` имеются два параметра типа, обозначающие типы модели и идентификатора строк таблицы. При манипулировании таблицами модель всегда относится к подтипу `TableModel`, а идентификатор — к типу

Integer. (Когда-нибудь остальные компоненты Swing будут также поддерживать фильтрацию строк, как и в таблицах. Например, для отбора строк в компоненте JTree может потребоваться класс RowFilter<TreeModel, TreePath>.)

Фильтр строк таблицы должен реализовывать следующий метод:

```
public boolean include(RowFilter.Entry<? extends M,
                       ? extends I> entry)
```

В классе RowFilter.Entry предоставляются методы для получения модели, идентификатора строк таблицы и значения по заданному индексу. Таким образом, фильтрацию можно производить как по идентификатору строк таблицы, так и по их содержимому. Например, с помощью следующего фильтра отображается каждая вторая строка таблицы:

```
var filter = new RowFilter<TableModel, Integer>()
{
  public boolean include(Entry<? extends TableModel,
                         ? extends Integer> entry)
  {
    return entry.getIdentifier() % 2 == 0;
  }
};
```

Если же требуется отобрать только те планеты, которые содержат четное количество спутников, то вместо приведенного выше фильтра можно попытаться применить следующий фильтр:

```
((Integer) entry.getValue(MOONS_COLUMN)) % 2 == 0
```

В рассматриваемом здесь примере программы пользователю разрешается скрывать произвольные строки таблицы. Индексы скрытых строк сохраняются в наборе строк. А в фильтр строк включаются все строки таблицы, индекс которых отсутствует в данном наборе.

Такой механизм фильтрации не предназначен для применения фильтров, критерии которых изменяются с течением времени. В рассматриваемом здесь примере программы вызов приведенного ниже метода повторяется всякий раз, когда изменяется набор скрытых строк таблицы. Как только фильтр установлен, он сразу же применяется.

```
sorter.setRowFilter(filter);
```

11.1.3.8. Сокрытие и показ столбцов таблицы

Как было показано в предыдущем подразделе, строки таблицы можно отфильтровывать по их содержимому или идентификатору. Для сокрытия столбцов понадобится совершенно другой механизм.

Метод removeColumn() из класса JTable позволяет удалить столбец, определяемый параметром TableColumn, из представления таблицы, т.е. скрыть его от пользователя, оставив в составе модели таблицы. Ниже показано, каким образом конкретный объект, описывающий столбец таблицы, извлекается из модели таблицы по известному номеру, получаемому, например, с помощью метода getSelectedColumns().

```
TableColumnModel columnModel = table.getColumnModel();
TableColumn column = columnModel.getColumn(i);
table.removeColumn(column);
```

Если запомнить этот объект, то впоследствии его можно ввести обратно в модель таблицы следующим образом:

```
table.addColumn(column);
```

Этот метод добавляет столбец в конец таблицы. Если же столбец требуется расположить в каком-нибудь другом месте, то для его перемещения на это место следует вызвать метод `moveColumn()`.

Кроме того, создав объект типа `TableColumn`, можно сформировать новый столбец, который соответствует индексу столбца в модели таблицы, как показано ниже. Таким образом, в таблице можно создать несколько столбцов, которые будут представлять один и тот же столбец в модели.

```
table.addColumn(new TableColumn(modelColumnIndex));
```

В рассматриваемом здесь примере программы демонстрируется выбор и фильтрация строк и столбцов таблицы. Исходный код этой программы представлен в листинге 11.3.

Листинг 11.3. Исходный код из файла **tableRowColumn/PlanetTableFrame.java**

```
1   package tableRowColumn;
2
3   import java.awt.*;
4   import java.util.*;
5
6   import javax.swing.*;
7   import javax.swing.table.*;
8
9   /**
10   * Этот фрейм содержит таблицы с данными о планетах
11   */
12  public class PlanetTableFrame extends JFrame
13  {
14    private static final int DEFAULT_WIDTH = 600;
15    private static final int DEFAULT_HEIGHT = 500;
16
17    public static final int COLOR_COLUMN = 4;
18    public static final int IMAGE_COLUMN = 5;
19
20    private JTable table;
21    private HashSet<Integer> removedRowIndices;
22    private ArrayList<TableColumn> removedColumns;
23    private JCheckBoxMenuItem rowsItem;
24    private JCheckBoxMenuItem columnsItem;
25    private JCheckBoxMenuItem cellsItem;
26
27    private String[] columnNames = { "Planet", "Radius",
28            "Moons", "Gaseous", "Color", "Image" };
29
30    private Object[][] cells = {
```

```
31        { "Mercury", 2440.0, 0, false, Color.YELLOW,
32          new ImageIcon(getClass()
33             .getResource("Mercury.gif")) },
34        { "Venus", 6052.0, 0, false, Color.YELLOW,
35          new ImageIcon(getClass()
36             .getResource("Venus.gif")) },
37        { "Earth", 6378.0, 1, false, Color.BLUE,
38          new ImageIcon(getClass()
39             .getResource("Earth.gif")) },
40        { "Mars", 3397.0, 2, false, Color.RED,
41          new ImageIcon(getClass()
42             .getResource("Mars.gif")) },
43        { "Jupiter", 71492.0, 16, true, Color.ORANGE,
44          new ImageIcon(getClass()
45             .getResource("Jupiter.gif")) },
46        { "Saturn", 60268.0, 18, true, Color.ORANGE,
47          new ImageIcon(getClass()
48             .getResource("Saturn.gif")) },
50        { "Uranus", 25559.0, 17, true, Color.BLUE,
51          new ImageIcon(getClass()
52             .getResource("Uranus.gif")) },
53        { "Neptune", 24766.0, 8, true, Color.BLUE,
54          new ImageIcon(getClass()
55             .getResource("Neptune.gif")) },
56        { "Pluto", 1137.0, 1, false, Color.BLACK,
57          new ImageIcon(getClass()
58             .getResource("Pluto.gif")) } };
59
60    public PlanetTableFrame()
61    {
62      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
63
64      var model =
65          new DefaultTableModel(cells, columnNames)
66      {
67        public Class<?> getColumnClass(int c)
68        {
69          return cells[0][c].getClass();
70        }
71      };
72
73      table = new JTable(model);
74
75      table.setRowHeight(100);
76      table.getColumnModel().getColumn(COLOR_COLUMN)
77                            .setMinWidth(250);
78      table.getColumnModel().getColumn(IMAGE_COLUMN)
79                            .setMinWidth(100);
80
81      var sorter = new TableRowSorter<>(model);
82      table.setRowSorter(sorter);
83      sorter.setComparator(COLOR_COLUMN,
84            Comparator.comparing(Color::getBlue)
85                      .thenComparing(Color::getGreen)
86                      .thenComparing(Color::getRed));
87      sorter.setSortable(IMAGE_COLUMN, false);
```

```
88      add(new JScrollPane(table), BorderLayout.CENTER);
89
90      removedRowIndices = new HashSet<>();
91      removedColumns = new ArrayList<>();
92
93      var filter = new RowFilter<TableModel, Integer>()
94      {
95        public boolean include(Entry<? extends TableModel,
96                               ? extends Integer> entry)
97        {
98          return !removedRowIndices
99                 .contains(entry.getIdentifier());
100       }
101     };
102
103     // создать меню
104
105     var menuBar = new JMenuBar();
106     setJMenuBar(menuBar);
107
108     var selectionMenu = new JMenu("Selection");
109     menuBar.add(selectionMenu);
110
111     rowsItem = new JCheckBoxMenuItem("Rows");
112     columnsItem = new JCheckBoxMenuItem("Columns");
113     cellsItem = new JCheckBoxMenuItem("Cells");
114
115     rowsItem.setSelected(
116              table.getRowSelectionAllowed());
117     columnsItem.setSelected(
118                table.getColumnSelectionAllowed());
119     cellsItem.setSelected(
120               table.getCellSelectionEnabled());
121
122     rowsItem.addActionListener(event ->
123       {
124         table.clearSelection();
125         table.setRowSelectionAllowed(
126               rowsItem.isSelected());
127         updateCheckboxMenuItems();
128       });
129
130     columnsItem.addActionListener(event ->
131       {
132         table.clearSelection();
133         table.setColumnSelectionAllowed(
134               columnsItem.isSelected());
135         updateCheckboxMenuItems();
136       });
137     selectionMenu.add(columnsItem);
138
139     cellsItem.addActionListener(event ->
140       {
141         table.clearSelection();
142         table.setCellSelectionEnabled(
143               cellsItem.isSelected());
```

```
144         updateCheckboxMenuItems();
145       });
146     selectionMenu.add(cellsItem);
147
148     var tableMenu = new JMenu("Edit");
149     menuBar.add(tableMenu);
150
151     var hideColumnsItem = new JMenuItem("Hide Columns");
152     hideColumnsItem.addActionListener(event ->
153       {
154         int[] selected = table.getSelectedColumns();
155         TableColumnModel columnModel =
156                         table.getColumnModel();
157
158         // удалить столбцы из представления таблицы,
159         // начиная с последнего индекса, но не
160         // затрагивая номера столбцов
161
162         for (int i = selected.length - 1; i >= 0; i--)
163         {
164           TableColumn column =
165                columnModel.getColumn(selected[i]);
166           table.removeColumn(column);
167
168           // сохранить удаленные столбцы для отображения
169
170           removedColumns.add(column);
171         }
172       });
173     tableMenu.add(hideColumnsItem);
174
175     var showColumnsItem = new JMenuItem("Show Columns");
176     showColumnsItem.addActionListener(event ->
177       {
178         // восстановить все удаленные столбцы
179         for (TableColumn tc : removedColumns)
180           table.addColumn(tc);
181         removedColumns.clear();
182       });
183     tableMenu.add(showColumnsItem);
184
185     var hideRowsItem = new JMenuItem("Hide Rows");
186     hideRowsItem.addActionListener(event ->
187       {
188         int[] selected = table.getSelectedRows();
189         for (int i : selected)
190           removedRowIndices.add(
191                  table.convertRowIndexToModel(i));
192         sorter.setRowFilter(filter);
193       });
194     tableMenu.add(hideRowsItem);
195
196     var showRowsItem = new JMenuItem("Show Rows");
197     showRowsItem.addActionListener(event ->
198       {
199         removedRowIndices.clear();
```

```
200         sorter.setRowFilter(filter);
201       });
202     tableMenu.add(showRowsItem);
203
204     var printSelectionItem =
205         new JMenuItem("Print Selection");
206     printSelectionItem.addActionListener(event ->
207       {
208         int[] selected = table.getSelectedRows();
209         System.out.println("Selected rows:
210                   " + Arrays.toString(selected));
211         selected = table.getSelectedColumns();
212         System.out.println("Selected columns:
213                   " + Arrays.toString(selected));
214       });
215     tableMenu.add(printSelectionItem);
216   }
217
218   private void updateCheckboxMenuItems()
219   {
220     rowsItem.setSelected(
221              table.getRowSelectionAllowed());
222     columnsItem.setSelected(
223                table.getColumnSelectionAllowed());
224     cellsItem.setSelected(
225               table.getCellSelectionEnabled());
226   }
227 }
```

javax.swing.table.TableModel 1.2

- **Class getColumnClass(int *columnIndex*)**

  Возвращает класс для определения типа значений в указанном столбце. Эти сведения используются для сортировки и воспроизведения значений в указанном столбце.

javax.swing.JTable 1.2

- **TableColumnModel getColumnModel()**

  Получает модель столбца, описывающую расположение столбцов в таблице.

- **void setAutoResizeMode(int *mode*)**

  Устанавливает режим автоматического изменения размеров столбцов таблицы.

  Параметры: ***mode*** Одно из значений следующих констант:

  AUTO_RESIZE_OFF,
  AUTO_RESIZE_NEXT_COLUMN,
  AUTO_RESIZE_SUBSEQUENT_COLUMNS,
  AUTO_RESIZE_LAST_COLUMN,
  AUTO_RESIZE_ALL_COLUMNS

javax.swing.JTable 1.2 *(окончание)*

- **int getRowMargin()**
- **void setRowMargin(int *margin*)**

  Получают или устанавливают ширину интервала между ячейками смежных строк таблицы.
- **int getRowHeight()**
- **void setRowHeight(int *height*)**

  Получают или устанавливают высоту всех строк таблицы по умолчанию.
- **int getRowHeight(int *row*)**
- **void setRowHeight(int *row*, int *height*)**

  Получают или устанавливают высоту указанной строки таблицы.
- **ListSelectionModel getSelectionModel()**

  Возвращает модель выбора списка. Эта модель требуется для того, чтобы сделать выбор между строкой, столбцом и ячейкой.
- **boolean getRowSelectionAllowed()**
- **void setRowSelectionAllowed(boolean *b*)**

  Получают или устанавливают свойство **rowSelectionAllowed**. Если оно принимает логическое значение **true**, то строки выбираются из таблицы, если щелкнуть на ее ячейках.
- **boolean getColumnSelectionAllowed()**
- **void setColumnSelectionAllowed(boolean *b*)**

  Получают или устанавливают свойство **columnSelectionAllowed**. Если оно принимает логическое значение **true**, то столбцы выбираются из таблицы, если щелкнуть кнопкой мыши на ее ячейках.
- **boolean getCellSelectionEnabled()**

  Возвращает логическое значение **true**, если оба свойства, **rowSelectionAllowed** и **columnSelectionAllowed**, принимают логическое значение **true**.
- **void setCellSelectionEnabled(boolean *b*)**

  Присваивает обоим свойствам, **rowSelectionAllowed** и **columnSelectionAllowed**, значение параметра *b*.
- **void addColumn(TableColumn *column*)**

  Добавляет столбец, становящийся последним в представлении таблицы.
- **void moveColumn(int *from*, int *to*)**

  Перемещает столбец, находящийся в таблице по индексу *from*, на новое место по индексу *to*. Эта операция затрагивает только представление таблицы.
- **void removeColumn(TableColumn *column*)**

  Удаляет указанный столбец из представления таблицы.
- **int convertRowIndexToModel(int *index*) 6**
- **int convertColumnIndexToModel(int *index*)**

  Возвращают индекс строки или столбца в модели по указанному индексу. Значение возвращаемого индекса отличается от значения параметра *index*, если производится сортировка или фильтрация строк таблицы или же если столбцы перемещаются или удаляются из таблицы.
- **void setRowSorter(RowSorter<? extends TableModel> *sorter*)**

  Устанавливает средство сортировки строк таблицы.

javax.swing.table.TableColumnModel 1.2

- **TableColumn getColumn(int *index*)**
  Возвращает объект, описывающий столбец по указанному индексу в представлении таблицы.

javax.swing.table.TableColumn 1.2

- **TableColumn(int *modelColumnIndex*)**
  Создает столбец таблицы для представления столбца в модели таблицы по указанному индексу.
- **void setPreferredWidth(int *width*)**
- **void setMinWidth(int *width*)**
- **void setMaxWidth(int *width*)**
  Задают предпочтительную, минимальную или максимальную ширину столбца равной значению параметра ***width***.
- **void setWidth(int *width*)**
  Задает конкретную ширину столбца равной значению параметра ***width***.
- **void setResizable(boolean *b*)**
  Если параметр ***b*** принимает логическое значение **true**, то допускается изменять размеры столбца.

javax.swing.ListSelectionModel 1.2

- **void setSelectionMode(int *mode*)**
  Задает режим выбора.
  Параметры: ***mode*** Одно из значений следующих констант: **SINGLE_SELECTION**, **SINGLE_INTERVAL_SELECTION** или **MULTIPLE_INTERVAL_SELECTION**

javax.swing.DefaultRowSorter<M, I> 6

- **void setComparator(int *column*, Comparator<?> *comparator*)**
  Устанавливает средство для сравнения со значением в указанном столбце.
- **void setSortable(int *column*, boolean *enabled*)**
  Разрешает или запрещает сортировку для указанного столбца.
- **void setRowFilter(RowFilter<? super M,? super I> *filter*)**
  Устанавливает фильтр строк таблицы.

javax.swing.table.TableRowSorter<M extends TableModel> 6

- **void setStringConverter(TableStringConverter *stringConverter*)**
  Устанавливает преобразователь для сортировки и фильтрации строк таблицы.

javax.swing.table.TableStringConverter<M extends TableModel> 6

- **abstract String toString(TableModel *model*, int *row*, int *column*)**
  Преобразует в символьную строку значение из указанного места в модели таблицы. Этот метод можно переопределить.

javax.swing.RowFilter<M, I> 6

- **boolean include(RowFilter.Entry<? extends M,? extends I> *entry*)**
  Определяет строки таблицы, которые остались после фильтрации. Этот метод можно переопределить.
- **static <M,I> RowFilter<M,I> numberFilter(RowFilter.ComparisonType *type*, Number *number*, int...*indices*)**
- **static <M,I> RowFilter<M,I> dateFilter(RowFilter.ComparisonType *type*, Date *date*, int... *indices*)**
  Возвращают фильтр, включающий строки, которые содержат значения, совпадающие со сравниваемым числом или датой. В качестве вида сравнения можно указать значение одной из следующих констант: **EQUAL**, **NOT_EQUAL**, **AFTER** или **BEFORE**. Если заданы любые индексы столбцов в модели, то поиск будет производиться только в этих столбцах, а иначе — во всех столбцах. Для фильтрации чисел класс, определяющий тип значений в ячейках таблицы, должен совпадать с классом параметра *number*.
- **static <M,I> RowFilter<M,I> regexFilter(String *regex*, int... *indices*)**
  Возвращает фильтр, включающий строки, которые содержат значения, совпадающие со сравниваемым регулярным выражением. Если заданы любые индексы столбцов в модели, то поиск будет производиться только в этих столбцах, а иначе — во всех столбцах. Следует, однако, иметь в виду, что метод **getStringValue()** из класса **RowFilter.Entry** возвращает совпавшую символьную строку.
- **static <M,I> RowFilter<M,I> andFilter(Iterable<? extends RowFilter<? super M,? super I>> *filters*)**
- **static <M,I> RowFilter<M,I> orFilter(Iterable<? extends RowFilter<? super M,? super I>> *filters*)**
  Возвращают фильтр, включающий записи, входящие во все фильтры или хотя бы в один из фильтров.
- **static <M,I> RowFilter<M,I> notFilter(RowFilter<M,I> *filter*)**
  Возвращает фильтр, включающий записи, не входящие в указанный фильтр.

javax.swing.RowFilter.Entry<M, I> 6

- **I getIdentifier()**
  Возвращает идентификатор данной записи в строке таблицы.
- **M getModel()**
  Возвращает модель данной записи в строке таблицы.
- **Object getValue(int *index*)**
  Возвращает значение, хранящееся по указанному индексу в данной строке таблицы.
- **int getValueCount()**
  Возвращает количество значений, хранящихся в данной строке таблицы.
- **String getStringValue(int *index*)**
  Возвращает значение, хранящееся по указанному индексу в данной строке таблицы и преобразованное в символьную строку. Средство сортировки строк типа **TableRowSorter** создает записи, для которых в данном методе вызывается преобразователь отсортированных результатов в символьные строки.

11.1.4. Воспроизведение и редактирование ячеек

Как было показано в подразделе 11.1.3.2, тип столбца определяет способ воспроизведения ячеек таблицы. По умолчанию для типов `Boolean` и `Icon` предоставляются средства воспроизведения флажков или изображений. Для всех остальных типов приходится самостоятельно устанавливать специальное средство воспроизведения.

11.1.4.1. Воспроизведение ячеек таблицы

Средства воспроизведения ячеек в таблице подобны упоминавшимся ранее средствам воспроизведения ячеек в списке. Они реализуют интерфейс `TableCellRenderer` с приведенным ниже единственным методом.

```
Component getTableCellRendererComponent(
          JTable table, Object value, boolean isSelected,
          boolean hasFocus, int row, int column)
```

Этот метод вызывается всякий раз, когда требуется снова воспроизвести таблицу. Он возвращает компонент, метод `paint()` которого служит для отображения содержимого ячейки.

В таблице, представленной на рис. 11.8, содержатся ячейки типа `Color`. Средство воспроизведения возвращает панель с цветом фона в виде объекта цвета, хранящегося в данной ячейке. Требующийся цвет передается в качестве параметра `value` методу, устанавливающему цвет фона:

```
class ColorTableCellRenderer extends JPanel
      implements TableCellRenderer
{
  public Component getTableCellRendererComponent(
         JTable table, Object value, boolean isSelected,
         boolean hasFocus, int row, int column)
  {
```

```
    setBackground((Color) value);
    if (hasFocus)
      setBorder(UIManager.getBorder(
                "Table.focusCellHighlightBorder"));
    else
      setBorder(null);
    return this;
  }
}
```

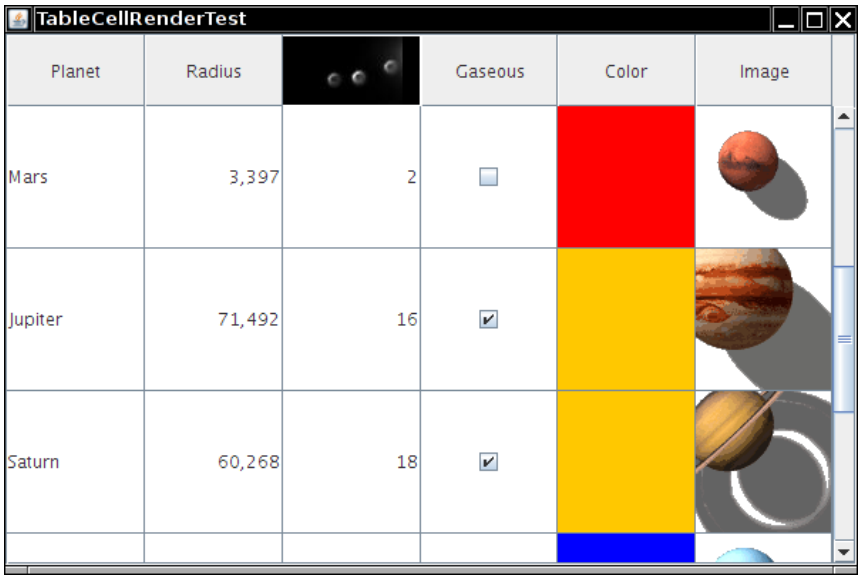


Рис. 11.8. Таблица со средствами воспроизведения ячеек

Нетрудно заметить, что при получении ячейкой фокуса ввода средство воспроизведения устанавливает рамку. (Для установки нужной рамки запрашивается объект типа `UIManager`. А для выбора подходящего ключа поиска приходится обращаться к исходному коду класса `DefaultTableCellRenderer`.)

СОВЕТ. Если конкретное средство воспроизведения должно просто выводить текстовую строку или пиктограмму, его можно построить в виде подкласса, расширяющего класс **`DefaultTableCellRenderer`**. В таком случае средства, реализованные в суперклассе, возьмут на себя ответственность за воспроизведение ячейки в состоянии выбора или обладания фокусом ввода.

Таблице необходимо каким-то образом указать, что для воспроизведения всех объектов типа `Color` следует использовать данное средство. Для этого предусмотрен метод `setDefaultRenderer()` из класса `JTable`. Ему передаются объект типа `Class` и требующееся средство воспроизведения следующим образом:

```
table.setDefaultRenderer(Color.class,
                         new ColorTableCellRenderer());
```

В итоге указанное средство будет использоваться для воспроизведения всех объектов данного типа. Если средство воспроизведения требуется выбрать по какому-нибудь другому критерию, придется создать подкласс, производный от класса `JTable`, а также переопределить метод `getCellRenderer()`.

11.1.4.2. Воспроизведение заголовков

Чтобы воспроизвести пиктограмму или иное изображение в заголовке столбца таблицы, необходимо установить соответствующее значение для этого заголовка следующим образом:

```
moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));
```

Но ведь заголовок столбца таблицы не настолько логически развит, чтобы самостоятельно выбирать подходящее средство воспроизведения по указанному значению. Поэтому данное средство придется установить вручную. Например, для того чтобы показать пиктограмму в заголовке столбца, необходимо вызвать следующий метод:

```
moonColumn.setHeaderRenderer(table.getDefaultRenderer(
                             ImageIcon.class));
```

11.1.4.3. Редактирование ячеек таблицы

Чтобы разрешить редактирование ячеек таблицы, следует получить из модели таблицы сведения о том, какие именно ячейки можно редактировать. Для этой цели служит метод `isCellEditable()`. Чаще всего редактируемыми объявляются не отдельные ячейки, а целые столбцы. Например, в приведенном ниже фрагменте кода разрешается редактирование данных в четырех столбцах.

```
public boolean isCellEditable(int r, int c)
{
  return c == PLANET_COLUMN || c == MOONS_COLUMN
         || c == GASEOUS_COLUMN || c ==   COLOR_COLUMN;
}
```

НА ЗАМЕТКУ! В классе **`AbstractTableModel`** имеется метод **`isCellEditable()`**, возвращающий логическое значение **`false`**, а в классе **`DefaultTableModel`** этот метод переопределен и по умолчанию возвращает логическое значение **`true`**.

В примере программы, исходный код которой представлен в листингах 11.4–11.7, можно устанавливать и сбрасывать флажки непосредственно в ячейках столбца `Gaseous` таблицы с данными о планетах. В этой программе можно также выбирать требующееся количество спутников планет из комбинированных списков в ячейках столбца `Moons`, как показано на рис. 11.9. Ниже поясняется, как организовать редактор ячеек таблицы на основании комбинированного списка. Наконец, если щелкнуть на любой ячейке в первом столбце `Planet` рассматриваемой здесь таблицы, эта ячейка получит фокус ввода, а следовательно, в нее можно ввести или отредактировать название планеты.

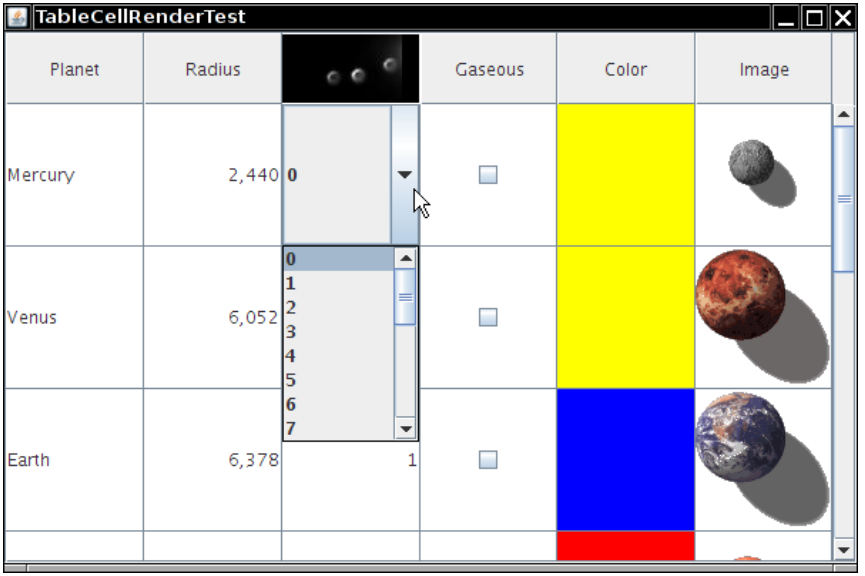


Рис. 11.9. Редактирование ячеек таблицы

В данном примере программы демонстрируется применение трех вариантов класса `DefaultCellEditor` для редактирования ячеек типа `JTextField`, `JCheckBox` и `JComboBox`. Класс `JTable` автоматически устанавливает редактор флажков для ячеек типа `Boolean`, а также редактор текстовых полей для всех редактируемых ячеек, у которых отсутствует собственное средство воспроизведения. В текстовых полях имеется возможность редактировать символьные строки, которые получаются в результате вызова метода `toString()` со значением, возвращаемым методом `getValueAt()` из модели таблицы.

По завершении редактирования в результате вызова метода `getCellEditorValue()` отредактированное значение извлекается из соответствующего редактора. Этот метод должен возвратить значение правильного типа (т.е. того типа, который возвращается методом `getColumnType()` из модели).

Редактор ячеек таблицы на основании комбинированного списка придется создать вручную, потому что компоненту `JTable` неизвестно, какие именно значения могут подойти для отдельного вида ячейки. Так, в столбце `Moons` требуется организовать выбор любого значения в пределах от `0` до `20`. Ниже приведен соответствующий код для инициализации этими значениями комбинированного списка.

```
var moonCombo = new JComboBox();
for (int i = 0; i <= 20; i++)
  moonCombo.addItem(i);
```

Чтобы создать объект класса `DefaultCellEditor` для такого типа данных, необходимо передать комбинированный список конструктору этого класса следующим образом:

```
var moonEditor = new DefaultCellEditor(moonCombo);
```

Затем следует установить редактор, который, в отличие от средства воспроизведения цвета в столбце `Color`, не должен зависеть от *типа* объекта. Его совсем не обязательно использовать для всех объектов типа `Integer`, а достаточно установить только в отдельном столбце следующим образом:

```
moonColumn.setCellEditor(moonEditor);
```

11.1.4.4. Специальные редакторы

Запустите еще раз на выполнение рассматриваемую здесь программу и щелкните кнопкой мыши на ячейке столбца `Color`. На экране появится диалоговое окно *селектора цвета* планеты. Выберите нужный цвет и щелкните на кнопке OK. В итоге цвет ячейки изменится на выбранный (рис. 11.10).

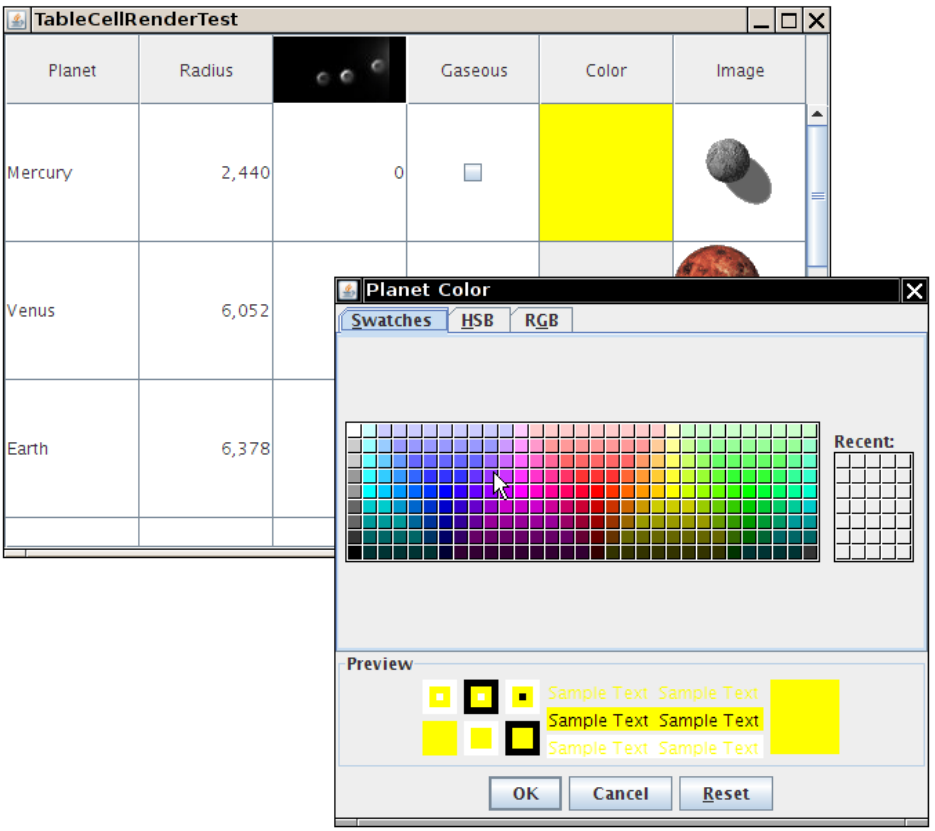


Рис. 11.10. Выбор цвета ячейки таблицы

Редактор цвета ячеек таблицы является не стандартным, а специальным, т.е. определяется и реализуется разработчиком прикладной программы. Для создания специального редактора ячеек таблицы следует реализовать интерфейс `TableCellEditor`. Впрочем, пользоваться этим интерфейсом не очень удобно, поэтому с версии Java 1.3 для обработки событий предоставляется класс `AbstractClassEditor`.

Метод `getTableCellEditorComponent()` из интерфейса `TableCellEditor` запрашивает компонент разрешения на воспроизведение ячейки таблицы. Он действует таким же образом, как и метод `getTableCellRendererComponent()` из интерфейса `TableCellRenderer`, за исключением того, что у него отсутствует параметр `focus`. Дело в том, что при редактировании ячейки таблицы предполагается, что эта ячейка обязательно имеет фокус ввода. На время редактирования компонент редактора временно *заменяет* средство воспроизведения. В данном случае возвращается пустая панель, для которой цвет не устанавливается. Это явно указывает пользователю на то, что в данный момент ячейка редактируется.

Далее требуется отобразить соответствующий редактор, как только пользователь щелкнет на ячейке таблицы. Компонент `JTable` вызывает редактор в ответ на соответствующее событие (например, щелчок кнопкой мыши). Для приема всех событий, которые могут инициировать редактирование, в классе `AbstractCellEditor` предусмотрен приведенный ниже метод. Но если переопределить этот метод таким образом, чтобы он возвращал логическое значение `false`, то компонент редактора не будет установлен в таблице.

```
public boolean isCellEditable(EventObject anEvent)
{
  return true;
}
```

После установки компонента редактора вызывается метод `shouldSelectCell()` — предположительно, с тем же самым событием. В этом методе инициализируется процесс редактирования, например, отображается окно внешнего редактора:

```
public boolean shouldSelectCell(EventObject anEvent)
{
  colorDialog.setVisible(true);
  return true;
}
```

Если пользователь откажется от редактирования ячейки таблицы, вызывается метод `cancelCellEditing()`, а если он щелкнет на другой ячейке таблицы — метод `stopCellEditing()`. В обоих случаях следует закрыть диалоговое окно соответствующего редактора. При вызове метода `stopCellEditing()` в таблице может появиться частично отредактированное значение. Если такое значение является допустимым, следует возвратить логическое значение `true`. Любое значение, выбранное в окне селектора цвета, будет допустимым, но при редактировании других данных следует убедиться в их достоверности.

Кроме того, из суперкласса следует вызвать методы, отвечающие за инициирование соответствующих событий. В противном случае редактирование не удастся отменить должным образом. В приведенном ниже фрагменте кода показано, каким образом отменяется редактирование.

```
public void cancelCellEditing()
{
  colorDialog.setVisible(false);
  super.cancelCellEditing();
}
```

Наконец, необходимо определить метод, возвращающий значение, получающееся в результате редактирования:

```
public Object getCellEditorValue()
{
  return colorChooser.getColor();
}
```

Таким образом, к специальному редактору ячеек таблицы предъявляются следующие требования.

1. Он должен расширять класс `AbstractCellEditor` и реализовывать интерфейс `TableCellEditor`.
2. В нем должен быть определен метод `getTableCellEditorComponent()`, предназначенный для предоставления компонента редактора. Это может быть фиктивный компонент, если редактор предполагается отображать в отдельном диалоговом окне, или же компонент для редактирования *непосредственно* в ячейке, как, например, комбинированный список или текстовое поле.
3. В нем должны быть определены методы `shouldSelectCell()`, `stopCellEditing()` и `cancelCellEditing()` для управления запуском, завершением и отменой процесса редактирования. Для уведомления обработчиков событий из суперкласса должны быть вызваны методы `stopCellEditing()` и `cancelCellEditing()`.
4. В нем должен быть определен метод `getCellEditorValue()`, возвращающий значение, получаемое в результате редактирования.

Наконец, когда пользователь завершит редактирование, следует вызвать метод `stopCellEditing()` или `cancelCellEditing()`. Так, при создании диалогового окна селектора цвета устанавливаются приведенные ниже методы обратного вызова, уведомляющие о подтверждении или отмене результатов редактирования и инициирующие соответствующие события.

```
colorDialog = JColorChooser.createDialog(
              null, "Planet Color", false, colorChooser,
              EventHandler.create(ActionListener.class,
                          this, "stopCellEditing"),
              EventHandler.create(ActionListener.class,
                          this, "cancelCellEditing"));
```

На этом реализация специального редактора завершается. Теперь вы знаете, как сделать ячейку таблицы редактируемой и установить ее редактор. Остается открытым лишь один вопрос: как обновить модель с учетом отредактированного значения? По завершении редактирования компонент `JTable` вызывает следующий метод из модели таблицы:

```
void setValueAt(Object value, int r, int c)
```

Для сохранения нового значения этот метод придется переопределить. Параметр `value` теперь будет обозначать объект, возвращаемый редактором ячейки таблицы. В определении редактора ячейки таблицы задается тип объекта, который возвращается методом `getCellEditorValue()`. Так, при использовании

класса `DefaultCellEditor` возможны три варианта указания типа для этого значения. В частности, для редактора ячейки в виде флажка это тип `boolean`, для текстового поля — тип `String`, а для комбинированного списка — тип объекта, выбираемого пользователем.

Если объект `value` имеет другой тип, его необходимо привести к нужному типу. Чаще всего это происходит при редактировании чисел в текстовом поле. В данном примере комбинированный список содержит объекты типа `Integer`, поэтому никакого приведения типов не требуется.

Листинг 11.4. Исходный код из файла **tableCellRender/TableCellRenderFrame.java**

```
1  package tableCellRender;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.table.*;
6
7  /**
8   * Этот фрейм содержит таблицу с данными о планетах
9  */
10 public class TableCellRenderFrame extends JFrame
11 {
12   private static final int DEFAULT_WIDTH = 600;
13   private static final int DEFAULT_HEIGHT = 400;
14
15   public TableCellRenderFrame()
16   {
17     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19     var model = new PlanetTableModel();
20     var table = new JTable(model);
21     table.setRowSelectionAllowed(false);
22
23     // установить средства воспроизведения и
24     // редактирования ячеек таблицы
25     table.setDefaultRenderer(Color.class,
26                     new ColorTableCellRenderer());
27     table.setDefaultEditor(Color.class,
28                     new ColorTableCellEditor());
29     var moonCombo = new JComboBox<>();
30     for (int i = 0; i <= 20; i++)
31        moonCombo.addItem(i);
32
33     TableColumnModel columnModel =
34                      table.getColumnModel();
35     TableColumn moonColumn = columnModel.getColumn(
36                 PlanetTableModel.MOONS_COLUMN);
37     moonColumn.setCellEditor(
38                new DefaultCellEditor(moonCombo));
39     moonColumn.setHeaderRenderer(
40         table.getDefaultRenderer(ImageIcon.class));
41     moonColumn.setHeaderValue(new ImageIcon(getClass()
42                         .getResource("Moons.gif")));
```

```
43
44      // показать таблицу
45      table.setRowHeight(100);
46      add(new JScrollPane(table),
47          BorderLayout.CENTER);
48    }
49 }
```

Листинг 11.5. Исходный код из файла **tableCellRender/PlanetTableModel.java**

```
1  package tableCellRender;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.table.*;
6
7  /**
8   * Модель таблицы планет, определяющая значения,
9   * свойства воспроизведения и редактирования
10  * данных о планетах
11  */
12 public class PlanetTableModel extends AbstractTableModel
13 {
14   public static final int PLANET_COLUMN = 0;
15   public static final int MOONS_COLUMN = 2;
16   public static final int GASEOUS_COLUMN = 3;
17   public static final int COLOR_COLUMN = 4;
18
19   private Object[][] cells = {
20      { "Mercury", 2440.0, 0, false, Color.YELLOW,
21        new ImageIcon(getClass()
22           .getResource("Mercury.gif")) },
23      { "Venus", 6052.0, 0, false, Color.YELLOW,
24        new ImageIcon(getClass()
25           .getResource("Venus.gif")) },
26      { "Earth", 6378.0, 1, false, Color.BLUE,
27        new ImageIcon(getClass()
28           .getResource("Earth.gif")) },
29      { "Mars", 3397.0, 2, false, Color.RED,
30        new ImageIcon(getClass()
31           .getResource("Mars.gif")) },
32      { "Jupiter", 71492.0, 16, true, Color.ORANGE,
33        new ImageIcon(getClass()
34           .getResource("Jupiter.gif")) },
35      { "Saturn", 60268.0, 18, true, Color.ORANGE,
36       new ImageIcon(getClass()
37          .getResource("Saturn.gif")) },
38      { "Uranus", 25559.0, 17, true, Color.BLUE,
39        new ImageIcon(getClass()
40           .getResource("Uranus.gif")) },
41      { "Neptune", 24766.0, 8, true, Color.BLUE,
42        new ImageIcon(getClass()
43           .getResource("Neptune.gif")) },
44      { "Pluto", 1137.0, 1, false, Color.BLACK,
45        new ImageIcon(getClass()
```

```
46          .getResource("Pluto.gif")) } };
47
48    private String[] columnNames = { "Planet", "Radius",
49          "Moons", "Gaseous", "Color", "Image" };
50
51    public String getColumnName(int c)
52    {
53      return columnNames[c];
54    }
55
56    public Class<?> getColumnClass(int c)
57    {
58      return cells[0][c].getClass();
59    }
60
61    public int getColumnCount()
62    {
63      return cells[0].length;
64    }
65
66    public int getRowCount()
67    {
68      return cells.length;
69    }
70
71    public Object getValueAt(int r, int c)
72    {
73      return cells[r][c];
74    }
75
76    public void setValueAt(Object obj, int r, int c)
77    {
78      cells[r][c] = obj;
79    }
80    public boolean isCellEditable(int r, int c)
81    {
82      return c == PLANET_COLUMN || c == MOONS_COLUMN
83            || c == GASEOUS_COLUMN || c == COLOR_COLUMN;
84    }
85 }
```

Листинг 11.6. Исходный код из файла **tableCellRender/ColorTableCellRenderer.java**

```
1  package tableCellRender;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.table.*;
6
7  /**
8   * Это средство воспроизведения отображает цвет
9   * в виде панели с заданным цветом
10 */
```

```
11 public class ColorTableCellRenderer
12       extends JPanel implements TableCellRenderer
13 {
14   public Component getTableCellRendererComponent(
15          JTable table, Object value, boolean isSelected,
16          boolean hasFocus, int row, int column)
17   {
18     setBackground((Color) value);
19     if (hasFocus)
20       setBorder(UIManager.getBorder(
21                 "Table.focusCellHighlightBorder"));
22     else setBorder(null);
23     return this;
24   }
25 }
```

Листинг 11.7. Исходный код из файла **tableCellRender/ColorTableCellEditor.java**

```
1  package tableCellRender;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.beans.*;
6  import java.util.*;
7  import javax.swing.*;
8  import javax.swing.table.*;
9
10 /**
11  * Этот редактор открывает диалоговое окно селектора
12  * цвета для редактирования значения цвета в выбранной
13  * ячейке таблицы
14  */
15 public class ColorTableCellEditor
16       extends AbstractCellEditor
17       implements TableCellEditor
18 {
19   private JColorChooser colorChooser;
20   private JDialog colorDialog;
21   private JPanel panel;
22   public ColorTableCellEditor()
23   {
24     panel = new JPanel();
25     // подготовить диалоговое окно выбора цвета
26
27     colorChooser = new JColorChooser();
28     colorDialog = JColorChooser.createDialog(
29        null, "Planet Color", false, colorChooser,
30        EventHandler.create(ActionListener.class,
31                            this, "stopCellEditing"),
32        EventHandler.create(ActionListener.class,
33                            this, "cancelCellEditing"));
34   }
35
36   public Component getTableCellEditorComponent(
37          JTable table, Object value, boolean isSelected,
```

```
38          int row, int column)
39    {
40      // Именно здесь получается текущее значение цвета,
41      // сохраняемое в диалоговом окне на тот случай,
42      // если пользователь начнет редактирование
43      colorChooser.setColor((Color) value);
44      return panel;
45    }
46
47    public boolean shouldSelectCell(EventObject anEvent)
48    {
49      // начать редактирование
50      colorDialog.setVisible(true);
51
52      // уведомить вызывающую часть программы о том,
53      // что данную ячейку разрешается выбрать
54      return true;
55    }
56
57    public void cancelCellEditing()
58    {
59      // редактирование отменено - скрыть диалоговое окно
60      colorDialog.setVisible(false);
61      super.cancelCellEditing();
62    }
63
64    public boolean stopCellEditing()
65    {
66      // редактирование завершено - скрыть диалоговое окно
67      colorDialog.setVisible(false);
68      super.stopCellEditing();
69
70      // уведомить вызывающую часть программы о том, что
71      // данное значение цвета разрешается использовать
72      return true;
73    }
74    public Object getCellEditorValue()
75    {
76      return colorChooser.getColor();
77    }
78 }
```

javax.swing.JTable 1.2

- **TableCellRenderer getDefaultRenderer(Class<?> *type*)**
  Получает средство воспроизведения, выбираемое по умолчанию для указанного типа ячейки таблицы.
- **TableCellEditor getDefaultEditor(Class<?> *type*)**
  Получает редактор, выбираемый по умолчанию для указанного типа ячейки таблицы.

javax.swing.table.TableCellRenderer 1.2

- **Component getTableCellRendererComponent(JTable *table*, Object *value*, boolean *selected*, boolean *hasFocus*, int *row*, int *column*)**
  Возвращает компонент, метод **paint()** которого вызывается для воспроизведения ячейки таблицы.

Параметры:		
	table	Таблица, содержащая воспроизводимые ячейки
	value	Воспроизводимая ячейка
	selected	Принимает логическое значение **true**, если текущая ячейка выбрана
	hasFocus	Принимает логическое значение **true**, если текущая ячейка обладает фокусом ввода
	row, ***column***	Номер строки и столбца с воспроизводимой ячейкой

javax.swing.table.TableColumn 1.2

- **void setCellEditor(TableCellEditor *editor*)**
- **void setCellRenderer(TableCellRenderer *renderer*)**
  Устанавливают редактор и средство воспроизведения для всех ячеек в указанном столбце.
- **void setHeaderRenderer(TableCellRenderer *renderer*)**
  Устанавливает средство воспроизведения для ячейки с заголовком в указанном столбце.
- **void setHeaderValue(Object *value*)**
  Устанавливает значение, которое должно быть отображено в виде заголовка в данном столбце.

javax.swing.DefaultCellEditor 1.2

- **DefaultCellEditor(JComboBox *comboBox*)**
  Создает редактор, предоставляющий комбинированный список для выбора значений ячеек таблицы.

javax.swing.table.TableCellEditor 1.2

- **Component getTableCellEditorComponent(JTable *table*, Object *value*, boolean *selected*, int *row*, int *column*)**
  Возвращает компонент, метод **paint()** которого вызывается для воспроизведения ячейки таблицы.

Параметры:		
	table	Таблица, содержащая воспроизводимые ячейки
	value	Воспроизводимая ячейка
	selected	Принимает логическое значение **true**, если текущая ячейка выбрана
	row, ***column***	Номер строки и столбца с воспроизводимой ячейкой

javax.swing.CellEditor 1.2

- **boolean isCellEditable(EventObject *event*)**
  Возвращает логическое значение **true**, если событие пригодно для запуска процесса редактирования данной ячейки.
- **boolean shouldSelectCell(EventObject *anEvent*)**
  Запускает процесс редактирования. Как правило, возвращается логическое значение **true**, если редактируемая ячейка должна быть *выбрана*. Если же требуется, чтобы содержимое выбранной ячейки не изменилось в результате редактирования, следует возвратить логическое значение **false**.
- **void cancelCellEditing()**
  Отменяет процесс редактирования. Его результаты можно проигнорировать.
- **boolean stopCellEditing()**
  Останавливает процесс редактирования с намерением использовать его результаты. Возвращает логическое значение **true**, если значение, получаемое в результате редактирования, оказывается допустимым.
- **Object getCellEditorValue()**
  Возвращает результаты редактирования.
- **void addCellEditorListener(CellEditorListener *l*)**
- **void removeCellEditorListener(CellEditorListener *l*)**
  Вводят или удаляют обязательный приемник событий, наступающих при редактировании ячеек таблицы.

11.2. Деревья

Каждому пользователю компьютера, оперирующего иерархической файловой системой, знакомы древовидные представления файлов и каталогов. Это, конечно, лишь один из многих примеров применения древовидной структуры. В повседневной жизни такую же структуру образует система административно-территориального деления страны на штаты, области и города (рис. 11.11).

Программирующим на Java нередко приходится писать код для отображения подобных структур. И для этого в библиотеке Swing предусмотрен класс `JTree`. Вместе со вспомогательными классами он берет на себя все хлопоты по компоновке древовидной структуры и обработке запросов пользователей на разворачивание и свертывание узлов дерева. В этом разделе показывается, как пользоваться средствами, доступными в классе `JTree`, для построения древовидных структур.

Подобно другим сложным компонентам Swing, здесь рассматриваются только самые общие и наиболее распространенные приемы оперирования деревьями. Для углубленного изучения данного вопроса рекомендуется упоминавшаяся ранее дополнительная литература: книга *Graphic Java™: Mastering the JFC, Volume II: Swing, 3rd Edition* Дэвида М. Гери или же книга *Core Swing* Кима Топли.

Приступая к рассмотрению деревьев, необходимо сначала пояснить характерную для них терминологию (рис. 11.12). Прежде всего, дерево состоит из *узлов*. Каждый узел может быть *листом* (т.е. *краевым узлом*) или *иметь дочерние узлы*. У каждого узла, кроме корневого, имеется только один *родительский узел*.

А у всего дерева в целом имеется только один *корневой узел*. Ряд деревьев с собственными корневыми узлами называется *лесом*.

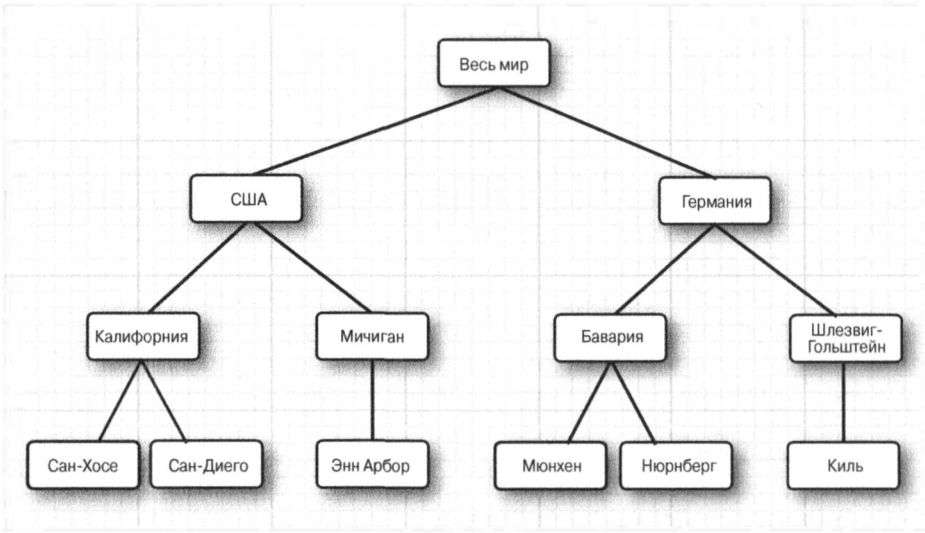


Рис. 11.11. Древовидная система административно-территориального деления страны на штаты, области и города

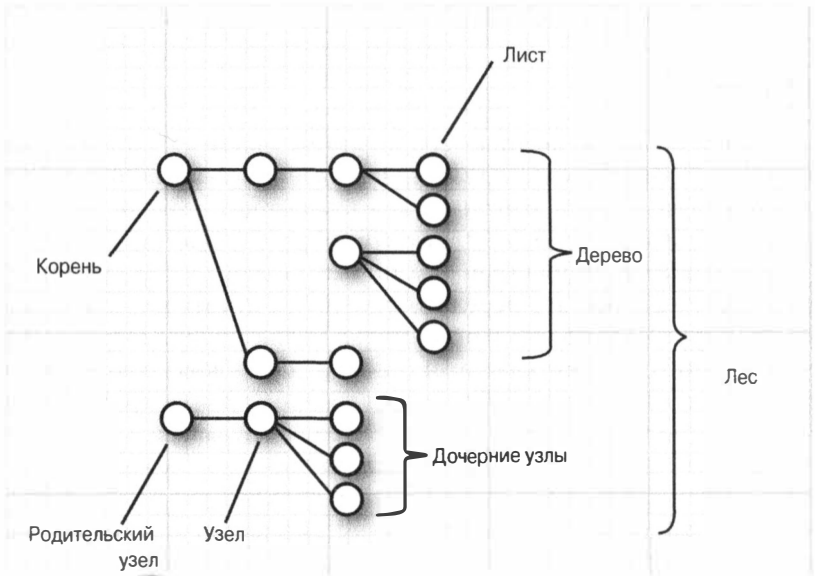


Рис. 11.12. Терминология, употребляемая для описания древовидных структур

11.2.1. Простые деревья

В рассматриваемом здесь первым несложном примере программы будет построено дерево лишь с несколькими узлами, как показано далее на рис. 11.14.

Подобно многим другим компонентам библиотеки Swing, программирующий на Java создает модель иерархических данных, а компонент `JTree` автоматически отображает их. Для создания объекта типа `JTree` его конструктору передается модель дерева следующим образом:

```
TreeModel model = . . .;
var tree = new JTree(model);
```

НА ЗАМЕТКУ! Некоторые конструкторы создают деревья из коллекции составляющих элементов, как показано ниже.

```
JTree(Object[] nodes)
JTree(Vector<?> nodes)
JTree(Hashtable<?, ?> nodes) // значения становятся узлами
```

Такие конструкторы практически бесполезны, потому что они создают только лес деревьев, каждое из которых содержит единственный узел. Третий конструктор из приведенных выше и вовсе не приносит никакой пользы, поскольку узлы передаются ему в произвольном порядке, который определяется хеш-кодами ключей.

Как же получить модель дерева? С помощью класса, реализующего интерфейс `TreeModel`, можно создать собственную модель. Такой способ будет рассмотрен далее в этой главе, а до тех пор воспользуемся моделью дерева типа `DefaultTreeModel`, предоставляемой в библиотеке Swing по умолчанию. Для построения используемой по умолчанию модели дерева необходимо предоставить конструктору корневой узел следующим образом:

```
TreeNode root = . . .;
var model = new DefaultTreeModel(root);
```

где `TreeNode` — еще один интерфейс. Используемую по умолчанию модель можно заполнить экземплярами любого класса, реализующего данный интерфейс. В данном случае применяется конкретный класс узлов дерева, называемый `DefaultMutableTreeNode` и входящий в состав библиотеки Swing. Как показано на рис. 11.13, этот класс реализует интерфейс `MutableTreeNode`, производный от интерфейса `TreeNode`.

Изменяемый по умолчанию узел дерева (т.е. экземпляр класса `DefaultMutableTreeNode`) содержит *пользовательский объект*. Эти объекты воспроизводятся для всех узлов. Если не задано специальное средство воспроизведения, то в дереве отображается символьная строка, получаемая в результате выполнения метода `toString()`.

В рассматриваемом здесь первом примере построения деревьев в качестве пользовательских объектов применяются символьные строки, хотя на практике древовидные структуры обычно заполняются более сложными объектами. Так, для отображения дерева каталогов в качестве его узлов имеет смысл использовать объекты типа `File`. Пользовательский объект можно указать сразу в конструкторе или в дальнейшем с помощью метода `setUserObject()`:

```
var node = new DefaultMutableTreeNode("Texas");
. . .
node.setUserObject("California");
```

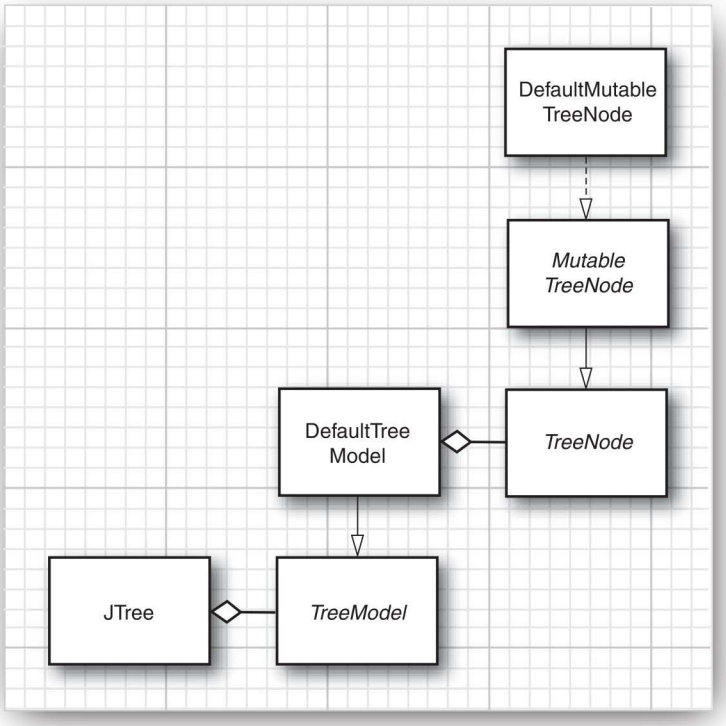


Рис. 11.13. Иерархия наследования интерфейсов и классов, используемых при построении деревьев

Затем между узлами следует установить отношения "родительский–дочерний". Для этого необходимо ввести дочерние узлы с помощью метода `add()`, начиная с корневого узла, как показано в приведенном ниже фрагменте кода. Получаемое в итоге простое дерево приведено на рис. 11.14.

```
var root = new DefaultMutableTreeNode("World");
var country = new DefaultMutableTreeNode("USA");
root.add(country);
var state = new DefaultMutableTreeNode("California");
country.add(state);
```

Подобным образом следует связать все узлы, а затем построить модель дерева типа `DeafultTreeNode` с корневым узлом, а на основе этой модели — само дерево с помощью компонента `JTree`:

```
var treeModel = new DefaultTreeModel(root);
var tree = new JTree(treeModel);
```

Можно поступить еще проще, передав корневой узел конструктору класса `JTree`, который автоматически создаст модель дерева по умолчанию, как показано ниже. Весь исходный код рассматриваемого здесь примера программы, демонстрирующей построение простого дерева, приведен в листинге 11.8.

```
var tree = new JTree(root);
```

Рис. 11.14. Простое дерево

Листинг 11.8. Исходный код из файла **tree/SimpleTreeFrame.java**

```
1  package tree;
2
3  import javax.swing.*;
4  import javax.swing.tree.*;
5
6  /**
7   * Этот фрейм содержит простое дерево, отображающее
8   * построенную вручную модель дерева
9  */
10 public class SimpleTreeFrame extends JFrame
11 {
12   private static final int DEFAULT_WIDTH = 300;
13   private static final int DEFAULT_HEIGHT = 200;
14
15   public SimpleTreeFrame()
16   {
17     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19     // подготовить данные для модели дерева
20
21     var root = new DefaultMutableTreeNode("World");
22     var country = new DefaultMutableTreeNode("USA");
23     root.add(country);
24     var state = new DefaultMutableTreeNode("California");
25     country.add(state);
26     var city = new DefaultMutableTreeNode("San Jose");
27     state.add(city);
28     city = new DefaultMutableTreeNode("Cupertino");
29     state.add(city);
30     state = new DefaultMutableTreeNode("Michigan");
31     country.add(state);
32     city = new DefaultMutableTreeNode("Ann Arbor");
33     state.add(city);
34     country = new DefaultMutableTreeNode("Germany");
35     root.add(country);
36     state = new DefaultMutableTreeNode("Schleswig-Holstein");
```

```
37      country.add(state);
38      city = new DefaultMutableTreeNode("Kiel");
39      state.add(city);
40
41      // построить дерево и разместить его на прокручиваемой панели:
42      JTree tree = new JTree(root);
43      add(new JScrollPane(tree));
44    }
45 }
```

Дерево, отображаемое при выполнении данной программы, показано на рис. 11.15. В рабочем окне программы будут видны только корневой узел и его дочерние узлы. Для развертывания поддеревьев следует щелкнуть мышью на пиктограмме кружка (*маркере* узла). Линия, соединяющаяся с кружком, направлена вправо, если поддерево свернуто, или вниз, если оно развернуто (рис. 11.16). Неизвестно, что имели в виду разработчики визуального стиля Metal, но, по-видимому, кружок, соединяемый с линией, обозначает дверную ручку, которую следует повернуть вниз по часовой стрелке, чтобы "открыть" поддерево.

Рис. 11.15. Исходное состояние отображаемого дерева

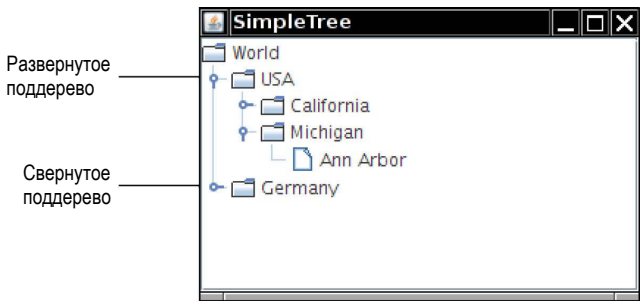


Рис. 11.16. Свернутые и развернутые поддеревья

НА ЗАМЕТКУ! Разумеется, внешний вид дерева зависит от выбранного визуального стиля. Приведенное выше описание относится к визуальному стилю Metal. В визуальных стилях Windows и Motif для обозначения свернутого и развернутого дерева употребляются знаки + и – (рис. 11.17).

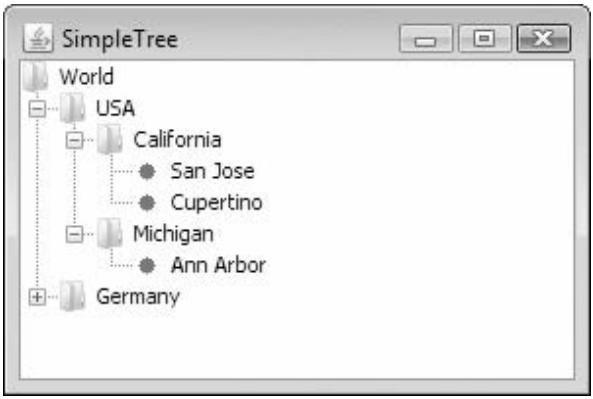


Рис. 11.17. Дерево, отображаемое в визуальном стиле Windows

Чтобы скрыть линии, связывающие родительские и дочерние узлы, как показано на рис. 11.18, необходимо указать значение `None` свойства `JTree.lineStyle` следующим образом:

```
tree.putClientProperty("JTree.lineStyle", "None");
```

Рис. 11.18. Дерево без соединительных линий между узлами

Для отображения этих линий следует указать значение `Angled` данного свойства, как показано ниже.

```
tree.putClientProperty("JTree.lineStyle", "Angled");
```

На рис. 11.19 показан еще один стиль представления структуры дерева с помощью горизонтальных линий. Такое дерево отображается с горизонтальными линиями, разделяющими только дочерние узлы корневого узла. Но трудно себе представить ситуацию, когда могла бы пригодиться такая древовидная структура.

По умолчанию у корневого узла отсутствует маркер для сворачивания всего дерева. Если требуется этот маркер, то следует вызвать приведенный ниже метод. На рис. 11.20 представлен результат вызова данного метода. Теперь все дерево можно сворачивать и разворачивать полностью.

```
tree.setShowsRootHandles(true);
```

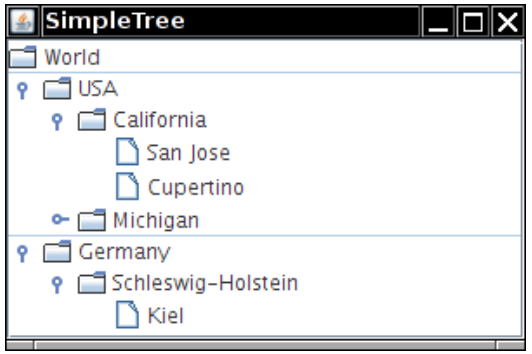


Рис. 11.19. Дерево с разделяющими горизонтальными линиями

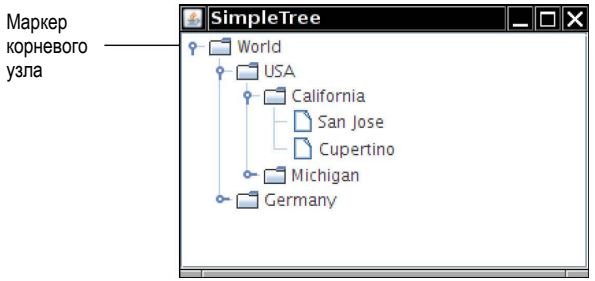


Рис. 11.20. Дерево с маркером корневого узла

Кроме того, корневой узел можно скрыть. Это может пригодиться, например, для отображения леса, т.е. ряда деревьев с собственными корневыми узлами. Все деревья следует объединить в дереве с общим корнем, а затем скрыть этот общий корень с помощью следующего метода:

```
tree.setRootVisible(false);
```

На рис. 11.21 приведен пример леса с двумя деревьями, имеющими корневые узлы `USA` и `Germany`, объединенные в одном дереве со скрытым корневым узлом.

Рис. 11.21. Лес

Перейдем теперь от корня к листьям дерева. Для их отображения служит пиктограмма листа бумаги, как показано на рис. 11.22.

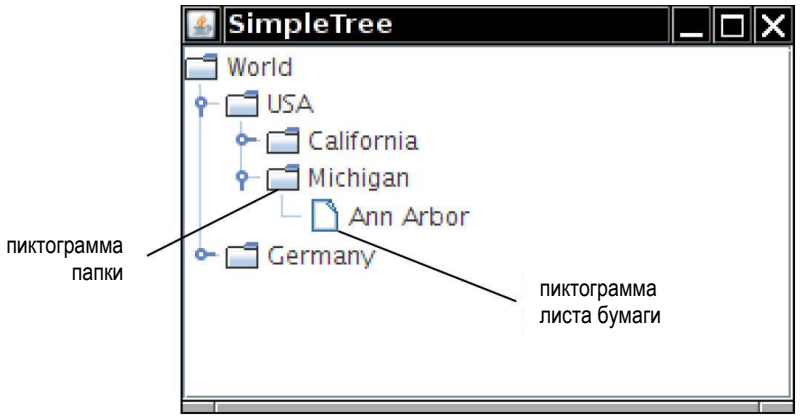


Рис. 11.22. Пиктограммы папок и листов дерева

Итак, каждый узел отображается отдельной пиктограммой. Для обозначения узлов дерева имеются три вида пиктограмм: лист бумаги, открытая и закрытая папки. Средству воспроизведения узлов должно быть известно, какой именно пиктограммой следует отображать каждый узел. По умолчанию решение принимается следующим образом: если метод `isLeaf()` возвращает логическое значение `true`, то используется пиктограмма листа бумаги, а иначе — пиктограмма папки.

Метод `isLeaf()` из класса `DefaultMutableTreeNode` возвращает логическое значение `true`, если у данного узла отсутствуют дочерние узлы. Таким образом, узлы дерева с дочерними узлами будут отображаться в виде папок, а узлы дерева без дочерних узлов — в виде листов бумаги.

Но такой способ обозначения узлов дерева подходит не для всех случаев. Например, при добавлении узла `Montana` в дерево для отображения штата Монтана без указания городов этот штат будет обозначен пиктограммой листа бумаги. Но при этом будет нарушен сам принцип представления дерева, по которому такие пиктограммы служат для обозначения городов.

Компоненту `JTree` неизвестно, какие именно узлы являются листами дерева, поэтому для выяснения этого факта он обращается к модели дерева. Если же узел без дочерних узлов не является листом дерева в принципе, то для определения листов можно выбрать другой критерий, например, обратиться к свойству узла, определяющему допустимость в нем дочерних узлов. Так, если для некоторого узла дочерние узлы недопустимы, то сначала необходимо вызвать следующий метод:

```
node.setAllowsChildren(false);
```

Затем следует запросить модель дерева, чтобы она выяснила с помощью свойства допустимости дочерних узлов, является ли узел листом дерева и следует ли отображать его пиктограммой листа бумаги. С этой целью необходимо вызвать метод `setAsksAllowsChildren()` из класса `DefaultTreeModel` следующим образом:

```
model.setAsksAllowsChildren(true);
```

В таком случае те узлы, в которых допускается наличие дочерних узлов, будут обозначены пиктограммами папок, а те узлы, в которых не допускается наличие дочерних узлов, — пиктограммами листов бумаги. С другой стороны, задавая в конструкторе дерева корневой узел, можно также указать на необходимость запрашивать свойство допустимости дочерних узлов, как показано ниже.

```
// те узлы, где не допускаются дочерние узлы,
// обозначаются пиктограммами листов бумаги:
var tree = new JTree(root, true);
```

javax.swing.JTree 1.2

- **JTree(TreeModel *model*)**
  Конструирует дерево из указанной модели.
- **JTree(TreeNode *root*)**
- **JTree(TreeNode *root*, boolean *asksAllowChildren*)**
  Конструируют дерево с заданной по умолчанию моделью, отображающей корневой узел и его дочерние узлы.

  Параметры: ***root*** Корневой узел
  asksAllowChildren Логическое значение **true** предписывает использовать свойство допустимости дочерних узлов, чтобы выяснить, является ли узел листом дерева
- **void setShowsRootHandles(boolean *b*)**
  Если параметр ***b*** принимает логическое значение **true**, то в корневом узле дерева отображается маркер свертывания.
- **void setRootVisible(boolean *b*)**
  Если параметр ***b*** принимает логическое значение **true**, то корневой узел отображается, а иначе он скрывается.

javax.swing.tree.TreeNode 1.2

- **boolean isLeaf()**
  Возвращает логическое значение **true**, если данный узел является листом.
- **boolean getAllowsChildren()**
  Возвращает логическое значение **true**, если данный узел может иметь дочерние узлы.

javax.swing.tree.MutableTreeNode 1.2

- **void setUserObject(Object *userObject*)**
  Задает пользовательский объект типа **userObject**, применяемый для воспроизведения узла дерева.

javax.swing.tree.TreeModel 1.2

- **boolean isLeaf(Object *node*)**
  Возвращает логическое значение **true**, если узел ***node*** следует отобразить как лист дерева.

javax.swing.tree.DefaultTreeModel 1.2

- **void setAsksAllowsChildren(boolean *b*)**
  Если параметр ***b*** принимает логическое значение **true**, то узлы отображаются как листья дерева, при условии, что метод **getAllowsChildren()** возвращает логическое значение **false**. В противном случае такой внешний вид узлов будет выбран, если метод **isLeaf()** возвратит логическое значение **true**.

javax.swing.tree.DefaultMutableTreeNode 1.2

- **DefaultMutableTreeNode(Object *userObject*)**
  Создает изменяемый узел дерева с указанным пользовательским объектом.
- **void add(MutableTreeNode *child*)**
  Вводит узел как последний дочерний узел в данном узле дерева.
- **void setAllowsChildren(boolean *b*)**
  Если параметр ***b*** принимает логическое значение **true**, в данный узел могут быть введены дочерние узлы.

javax.swing.JComponent 1.2

- **void putClientProperty(Object *key*, Object *value*)**
  Вводит указанную пару "ключ-значение" в небольшую таблицу, которой управляет каждый компонент. Этот "запасной" механизм используется в некоторых компонентах библиотеки Swing для хранения специальных свойств, определяющих их внешний вид.

11.2.1.1. Редактирование деревьев и путей к ним

В следующем примере программы демонстрируются способы редактирования деревьев. На рис. 11.23 показан пользовательский интерфейс данной программы, где для создания нового узла под названием New предусмотрены кнопки Add Sibling (Добавить равноправный узел) и Add Child (Добавить дочерний узел). Для удаления текущего выбранного узла предназначена кнопка Delete (Удалить).

Для реализации такого поведения следует найти текущий выбранный узел. В компоненте JTree предусмотрен интересный способ поиска такого узла по *пути к объекту*, называемому иначе *путем к дереву*. Такой путь начинается с корневого узла и состоит из последовательности дочерних узлов (рис. 11.24).

Рис. 11.23. Редактирование дерева

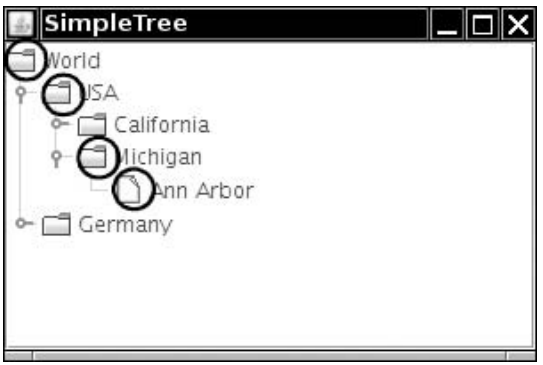


Рис. 11.24. Путь к дереву

Казалось бы, для поиска текущего узла лучше было бы воспользоваться интерфейсом `TreeNode` и методом `getParent()`. Но на самом деле компоненту `JTree` ничего неизвестно об интерфейсе `TreeNode`, поскольку он используется только для реализации в классе `DefaultTreeModel`, а не в интерфейсе `TreeModel`. Дерево может иметь модель, в узлах которой не реализуется интерфейс `TreeNode`. Например, в модели дерева с другими типами объектов могут вообще отсутствовать методы `getParent()` и `getChild()`. В таком случае для организации связей между узлами применяется модель дерева, хотя компоненту `JTree` ничего о них неизвестно. Поэтому в компоненте `JTree` предполагается всегда использовать полные пути к деревьям.

Класс `TreePath` управляет последовательностью ссылок на объекты типа `Object` (а не `TreeNode`) с помощью нескольких методов. Например, для получения ссылки на последний узел пути можно вызвать метод `getLastPathComponent()`. Для поиска текущего выбранного узла служит метод `getSelectionPath()` из класса `JTree`. Таким образом, зная путь к дереву (на основании объекта типа `TreePath`), можно получить ссылку на текущий выбранный узел следующим образом:

```
TreePath selectionPath = tree.getSelectionPath();
var selectedNode = (DefaultMutableTreeNode)
                   selectionPath.getLastPathComponent();
```

Такой запрос выполняется очень часто, поэтому для него разработан приведенный ниже служебный метод.

```
var selectedNode = (DefaultMutableTreeNode)
                   tree.getLastSelectedPathComponent();
```

Этот метод называется `getLastSelectedPathComponent()`, а не `getSelectedNode()`, поскольку самому дереву ничего неизвестно об узлах, а его модель оперирует только путями к объектам.

НА ЗАМЕТКУ! Кроме путей к деревьям, для описания узлов дерева используются методы из класса **`JTree`**, которые принимают или возвращают целочисленный индекс, обозначающий *позицию строки* (т.е. номер строки, начиная с нуля) для узла во внутреннем представлении дерева. Такие номера имеют только видимые узлы, причем номер строки может меняться при свертывании, развертывании или изменении дерева. Поэтому номера, обозначающие позиции строк, не рекомендуется применять для доступа к узлам. У каждого метода из класса **`JTree`**, предназначенного для работы с позициями строк, имеется эквивалентный метод, оперирующий путями к деревьям.

Получив выбранный узел, можно приступать к его редактированию. Но добавить к нему дочерние узлы нельзя, просто вызвав метод `add()`, как показано ниже.

```
selectedNode.add(newNode); // Нельзя!
```

Если изменяется структура узлов, то изменения вносятся в модель дерева, а связанное с ним представление об этом не уведомляется. Следовательно, такое уведомление необходимо послать самостоятельно. Но если для ввода нового узла вызвать метод `insertNodeInto()` из класса `DefaultTreeModel`, то такое уведомление будет отправлено представлению дерева автоматически:

```
model.insertNodeInto(newNode, selectedNode,
                     selectedNode.getChildCount());
```

Аналогичным образом применяется метод `removeNodeFromParent()` для удаления узла и уведомления об обновлении представления дерева:

```
model.removeNodeFromParent(selectedNode);
```

Для изменения пользовательского объекта, но с сохранением структуры узла необходимо вызвать метод `nodeChanged()` следующим образом:

```
model.nodeChanged(changedNode);
```

Автоматическое уведомление является основным достоинством модели типа `DefaultTreeModel`. При создании собственных моделей деревьев такое уведомление приходится организовывать самостоятельно. Более подробно этот вопрос обсуждается в упоминавшей ранее книге *Core Swing* Кима Топли.

ВНИМАНИЕ! В состав класса **`DeafultTreeModel`** входит метод **`reload()`**, полностью перезагружающий модель дерева. Но метод **`reload()`** не следует вызывать только для обновления дерева после внесения нескольких изменений в нее. Дело в том, что при таком обновлении дерева все узлы за пределами дочерних узлов корневого узла снова будут свернуты. Такое поведение дерева может оказаться неудобным для пользователей, вынуждая их развертывать дерево после каждого вносимого в него изменения.

Если представление получает уведомление об изменении структуры узлов дерева, оно обновляет отображение узлов, не развертывая их для просмотра вновь добавленных узлов. Например, добавление в рассматриваемом здесь примере программы нового дочернего узла в свернутые узлы произойдет незаметно для пользователя. В таком случае придется специально организовать развертывание родительских узлов для отображения введенного нового дочернего узла. Для этого из класса `JTree` можно вызвать метод `makeVisible()`, принимающий путь, ведущий к отображаемому на экране узлу.

Таким образом, для отображения вновь введенного узла придется сформировать путь к нему от корневого узла дерева. Для получения этого пути сначала следует вызвать метод `getPathToRoot()` из класса `DeafultTreeModel`. Он возвращает массив `TreeArray[]` для всех узлов (от текущего до корневого), который далее передается конструктору класса `TreePath`. В приведенном ниже фрагменте когда показано, каким образом вновь введенный узел делается видимым.

```
TreeNode[] nodes = model.getPathToRoot(newNode);
var path = new TreePath(nodes);
tree.makeVisible(path);
```

НА ЗАМЕТКУ! Любопытно, что класс **`DefaultTreeModel`** ведет себя так, как будто ему вообще ничего неизвестно о классе **`TreePath`**, хотя он и предназначен для взаимодействия с классом **`JTree`**. В то же время в классе **`JTree`** широко применяются пути и вообще не используются массивы объектов узлов.

Теперь допустим, что дерево находится на прокручиваемой панели. После развертывания дерева новый узел все еще может быть за пределами текущего окна просмотра. Для перехода к новому узлу следует вместо метода `makeVisible()` вызвать метод `scrollPathToVisible()`, как показано ниже. Этот метод развертывает все узлы, указанные в заданном пути, прокручивая содержимое окна вплоть до последнего узла в конце пути (рис. 11.25).

```
tree.scrollPathToVisible(path);
```

Рис. 11.25. Автоматическая прокрутка дерева на панели для просмотра нового узла

После двойного щелчка мышью откроется показанное на рис. 11.26 окно редактора ячеек, вызываемого по умолчанию. Для его реализации служит класс `DefaultCellEditor`. А для редактирования узлов с иными объектами, кроме символьных строк, можно установить другие редакторы ячеек. Это делается аналогично установке редакторов ячеек таблицы, как пояснялось ранее в данной главе.

Рис. 11.26. Используемый по умолчанию редактор ячеек дерева

В листинге 11.9 приведен весь исходный код рассматриваемого здесь примера программы редактирования отдельных узлов дерева. Запустите эту программу на выполнение, создайте несколько узлов и отредактируйте их, дважды щелкнув на имени узла. Убедитесь в том, что свернутые деревья разворачиваются и содержимое окна прокручивается для отображения нового дочернего узла в области просмотра.

Листинг 11.9. Исходный код из файла **treeEdit/TreeEditFrame.java**

```
1   package treeEdit;
2
3   import java.awt.*;
4
5   import javax.swing.*;
6   import javax.swing.tree.*;
7
8   /**
9    * Фрейм с деревом и кнопками для его редактирования
10  */
11  public class TreeEditFrame extends JFrame
12  {
13    private static final int DEFAULT_WIDTH = 400;
14    private static final int DEFAULT_HEIGHT = 200;
15
16    private DefaultTreeModel model;
17    private JTree tree;
18
19    public TreeEditFrame()
20    {
21      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
22
23      // построить дерево
24
25      TreeNode root = makeSampleTree();
26      model = new DefaultTreeModel(root);
27      tree = new JTree(model);
28      tree.setEditable(true);
29
30      // ввести прокручиваемую панель с деревом
```

```
31
32      var scrollPane = new JScrollPane(tree);
33      add(scrollPane, BorderLayout.CENTER);
34
35      makeButtons();
36    }
37
38    public TreeNode makeSampleTree()
39    {
40      var root = new DefaultMutableTreeNode("World");
41      var country = new DefaultMutableTreeNode("USA");
42      root.add(country);
43      var state = new DefaultMutableTreeNode(
44                      "California");
45      country.add(state);
46      var city = new DefaultMutableTreeNode("San Jose");
47      state.add(city);
48      city = new DefaultMutableTreeNode("San Diego");
49      state.add(city);
50      state = new DefaultMutableTreeNode("Michigan");
51      country.add(state);
52      city = new DefaultMutableTreeNode("Ann Arbor");
53      state.add(city);
54      country = new DefaultMutableTreeNode("Germany");
55      root.add(country);
56      state = new DefaultMutableTreeNode(
57                  "Schleswig-Holstein");
58      country.add(state);
59      city = new DefaultMutableTreeNode("Kiel");
60      state.add(city);
61      return root;
62    }
63
64    /**
65     * Создает кнопки для ввода родственных,
66     * дочерних узлов и их удаления
67     */
68    public void makeButtons()
69    {
70      var panel = new JPanel();
71      var addSiblingButton = new JButton("Add Sibling");
72      addSiblingButton.addActionListener(event ->
73        {
74          var selectedNode = (DefaultMutableTreeNode)
75                      tree.getLastSelectedPathComponent();
76
77          if (selectedNode == null) return;
78
79          var parent = (DefaultMutableTreeNode)
80                       selectedNode.getParent();
81
82          if (parent == null) return;
83
84          var newNode = new DefaultMutableTreeNode("New");
85
86          int selectedIndex =
87              parent.getIndex(selectedNode);
```

```
88          model.insertNodeInto(newNode, parent,
89                               selectedIndex + 1);
90
91          // отобразить теперь новый узел
92
93          TreeNode[] nodes = model.getPathToRoot(newNode);
94          var path = new TreePath(nodes);
95          tree.scrollPathToVisible(path);
96        });
97      panel.add(addSiblingButton);
98
99      var addChildButton = new JButton("Add Child");
100     addChildButton.addActionListener(event ->
101       {
102         var selectedNode = (DefaultMutableTreeNode)
103                    tree.getLastSelectedPathComponent();
104
105
106         if (selectedNode == null) return;
107
108         var newNode = new DefaultMutableTreeNode("New");
109         model.insertNodeInto(newNode, selectedNode,
110                         selectedNode.getChildCount());
111
112         // отобразить теперь новый узел
113
114         TreeNode[] nodes = model.getPathToRoot(newNode);
115         var path = new TreePath(nodes);
116         tree.scrollPathToVisible(path);
117       });
118     panel.add(addChildButton);
119
120     var deleteButton = new JButton("Delete");
121     deleteButton.addActionListener(event ->
122       {
123         var selectedNode = (DefaultMutableTreeNode)
124                    tree.getLastSelectedPathComponent();
125
126         if (selectedNode != null
127             && selectedNode.getParent() != null)
128           model.removeNodeFromParent(selectedNode);
129       });
130     panel.add(deleteButton);
131     add(panel, BorderLayout.SOUTH);
132   }
133 }
```

javax.swing.JTree 1.2

- **TreePath getSelectionPath()**

  Получает путь к текущему выбранному узлу дерева (или путь к первому выбранному узлу, если выбрано сразу несколько узлов). Возвращает пустое значение **null**, если ни один из узлов не выбран.

javax.swing.JTree 1.2 *(окончание)*

- **Object getLastSelectedPathComponent()**
  Получает объект, который представляет текущий выбранный узел дерева (или первый выбранный узел, если выбрано сразу несколько узлов). Возвращает пустое значение **null**, если ни один из узлов не выбран.
- **void makeVisible(TreePath *path*)**
  Развертывает все узлы дерева по заданному пути.
- **void scrollPathToVisible(TreePath *path*)**
  Развертывает все узлы дерева по заданному пути, и если дерево находится на прокручиваемой панели, то прокручивает его, чтобы обеспечить отображение последнего узла по заданному пути.

javax.swing.tree.TreePath 1.2

- **Object getLastPathComponent()**
  Получает последний объект по заданному пути, т.е. объект узла, указанный в пути к дереву.

javax.swing.tree.TreeNode 1.2

- **TreeNode getParent()**
  Возвращает родительский узел данного узла дерева.
- **TreeNode getChildAt(int *index*)**
  Ищет дочерний узел дерева по указанному индексу. Значение индекса должно находиться в пределах от **0** до **getChildCount() - 1**.
- **int getChildCount()**
  Возвращает количество дочерних узлов данного узла дерева.
- **Enumeration children()**
  Возвращает объект типа **Enumeration** для перебора всех дочерних узлов данного узла дерева.

javax.swing.tree.DefaultTreeModel 1.2

- **int *index*)**
  Вводит объект ***newChild*** в качестве дочернего узла в родительский узел ***parent*** по указанному индексу и уведомляет приемники событий от модели дерева.
- **void removeNodeFromParent(MutableTreeNode *node*)**
  Удаляет узел ***node*** из модели дерева и уведомляет приемники событий от этой модели.
- **void insertNodeInto(MutableTreeNode *newChild*, MutableTreeNode *parent*, void nodeChanged(TreeNode *node*)**
  Уведомляет приемники событий от модели дерева об изменениях в заданном узле ***node***.

javax.swing.tree.DefaultTreeModel 1.2 *(окончание)*

- **void nodesChanged(TreeNode *parent*, int[] *changedChildIndexes*)**
  Уведомляет приемники событий от модели дерева об изменениях во всех дочерних узлах родительского узла ***parent*** по указанным индексам.
- **void reload()**
  Перезагружает все узлы в модели дерева. Эту операцию следует выполнять только в тех случаях, когда узлы полностью изменились под внешним воздействием.

11.2.2. Перечисление узлов дерева

Иногда требуется найти узел дерева, начиная с корневого узла и перебирая все дочерние узлы до тех пор, пока не будет найден совпадающий узел. Для перебора узлов в классе `DefaultMutableTreeNode` предоставляется несколько удобных методов.

Методы `breadthFirstEnumeration()` и `depthFirstEnumeration()` возвращают объект типа `Enumeration`. У этого объекта имеется метод `nextElement()`, предназначенный для последовательного обращения ко всем дочерним узлам текущего узла. Если объект типа `Enumeration` получен с помощью метода `breadthFirstEnumeration()`, он представляет результаты обхода в ширину всех дочерних узлов текущего узла. А объект типа `Enumeration`, возвращаемый методом `depthFirstEnumeration()`, содержит результаты обхода всех дочерних узлов текущего узла в глубину. На рис. 11.27 схематически показаны оба способа обхода узлов дерева.

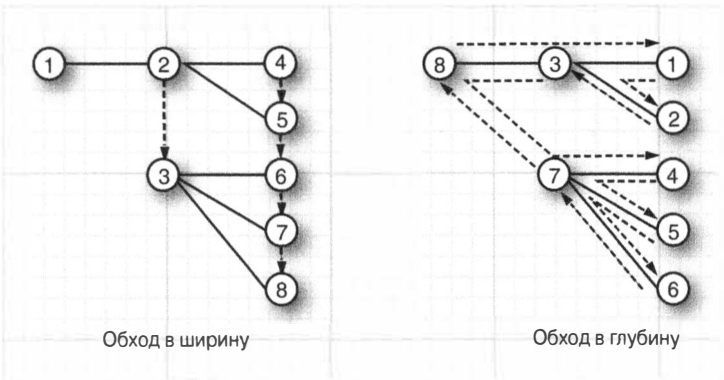


Рис. 11.27. Способы обхода узлов дерева

Обход узлов дерева в ширину выполняется по отдельным уровням: сначала корневой узел, затем все его дочерние узлы, после чего все дочерние узлы этих дочерних узлов и т.д. Воспроизвести результаты такого обхода совсем не трудно.

А вот обход узлов дерева в глубину похож на поиск выхода из лабиринта. В этом случае обход выполняется по какому-то одному пути до тех пор, пока не будет достигнут листовой узел. Затем происходит возврат назад и выбор

ближайшего нового пути, после чего обход продолжается по этому же пути до тех пор, пока не будет достигнут листовой узел, и т.д.

Такой способ иначе называется *обходом в обратном порядке*, поскольку в процессе поиска сначала посещаются дочерние, а затем родительские узлы. Именно поэтому метод `postOrderTraversal()` действует подобно методу `depthFirstTraversal()`. Следует отметить и метод `preOrderTraversal()`, который также предназначен для обхода в глубину, но в получаемых результатах родительские узлы предшествуют дочерним. Ниже приведен образец типичной реализации обхода дерева в ширину непосредственно в коде.

```
Enumeration breadthFirst = node.breadthFirstEnumeration();
while (breadthFirst.hasMoreElements())
    сделать что-нибудь с результатом вызова
    метода breadthFirst.nextElement();
```

Наконец, метод `pathFromAncestorEnumeration()` служит для поиска пути от родительского узла к заданному и для перечисления узлов по этому пути. Принцип его работы основывается на вызове метода `getParent()` для данного узла до тех пор, пока не будет найден заданный родительский узел. После этого предоставляется путь для обхода дерева в обратном порядке.

В следующем примере программы демонстрируется применение описанных выше методов для обхода дерева иерархического наследования классов. Как только в текстовом поле, расположенном в нижней части рабочего окна данной программы, будет указано имя конкретного класса, он будет введен в дерево иерархического наследования со всеми своими суперклассами, как показано на рис. 11.28.

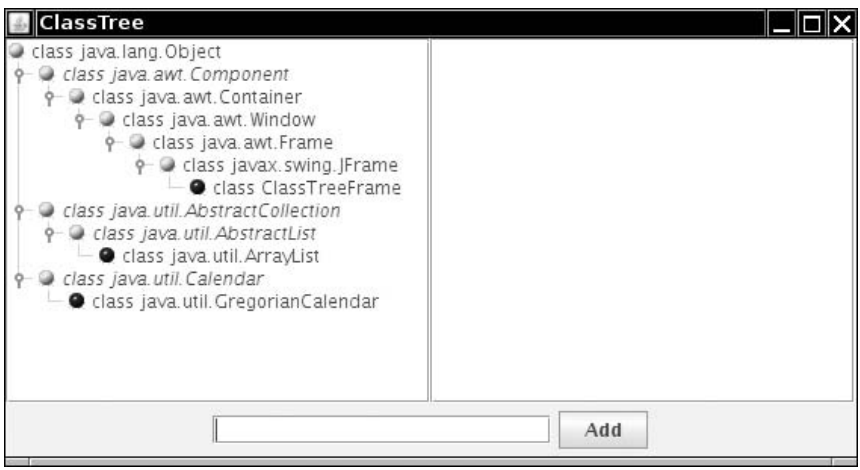


Рис. 11.28. Дерево иерархического наследования классов

В рассматриваемом здесь примере используется следующая важная особенность дерева: пользовательский объект узла может быть объектом произвольного типа. В данном примере узлы дерева представляют иерархическую структуру наследования классов, поэтому они относятся к одному типу `Class`. Во избежание дублирования следует организовать проверку наличия добавляемого класса в дереве. Это делается с помощью приведенного ниже метода.

```
public DefaultMutableTreeNode findUserObject(Object obj)
{
  Enumeration e = root.breadthFirstEnumeration();
  while (e.hasMoreElements())
  {
    DefaultMutableTreeNode node =
          (DefaultMutableTreeNode) e.nextElement();
    if (node.getUserObject().equals(obj))
      return node;
  }
  return null;
}
```

11.2.3. Воспроизведение узлов дерева

При разработке прикладных программ нередко возникает потребность изменить способ отображения узлов дерева, например, обозначить их другими пиктограммами или выделить их названия другим шрифтом. Подобные изменения можно вносить с помощью *средства воспроизведения ячеек дерева*. По умолчанию для воспроизведения узлов дерева в компоненте `JTree` применяется класс `DefaultTreeCellRenderer`, который расширяет класс `JLabel`. Этот класс формирует метку, состоящую из пиктограммы и метки узла.

НА ЗАМЕТКУ! Средство воспроизведения ячеек дерева не отображает маркеры свертывания и развертывания подчиненных деревьев. Они являются частью общего визуального стиля, поэтому изменять их не рекомендуется.

Специальную настройку воспроизведения узлов дерева можно произвести одним из трех способов.

- Заменить с помощью класса `DefaultTreeCellRenderer` пиктограммы, шрифты и цвета фона, применяемые во всех узлах дерева.
- Установить используемое по умолчанию средство воспроизведения, расширяющее класс `DefaultTreeCellRenderer`, а также применить разные пиктограммы, шрифты и цвета фона в отдельных узлах дерева.
- Установить средство воспроизведения, реализующее интерфейс `TreeCellRenderer`, чтобы воспроизводить в отдельных узлах дерева специально предусмотренные для них изображения.

Рассмотрим каждый из этих способов более подробно. Наиболее простой способ изменения пиктограммы, шрифта и цвета фона узлов состоит в том, чтобы построить объект типа `DefaultTreeCellRenderer` в качестве средства воспроизведения узлов дерева по умолчанию и установить его вместе с требуемыми пиктограммами в дереве, как показано ниже. Результат выполнения этих действий приведен на рис. 11.28, где в качестве пиктограмм узлов используются изображения шариков.

```
var renderer = new DefaultTreeCellRenderer();
renderer.setLeafIcon(new ImageIcon("blue-ball.gif"));
    // используется для листовых узлов
renderer.setClosedIcon(new ImageIcon("red-ball.gif"));
```

```
    // используется для свернутых узлов
renderer.setOpenIcon(new ImageIcon("yellow-ball.gif"));
    // используется для развернутых узлов
tree.setCellRenderer(renderer);
```

Изменять шрифт и цвет фона не рекомендуется, так как легко нарушить общий стиль оформления пользовательского интерфейса. Шрифт допускается изменять только для выделения отдельных узлов. Так, на рис. 11.28 курсивом выделены абстрактные классы.

Для изменения внешнего вида отдельных узлов следует установить специальное средство воспроизведения ячеек дерева. Данное средство во многом похоже на средство воспроизведения ячеек из списка, рассмотренное ранее в этой главе. У интерфейса `TreeCellRenderer` имеется следующий единственный метод `getTreeCellRendererComponent()`:

```
Component getTreeCellRendererComponent(JTree tree,
                    Object value, boolean selected,
                    boolean expanded, boolean leaf,
                    int row, boolean hasFocus)
```

Метод `getTreeCellRendererComponent()` из класса `DefaultTreeCellRenderer` возвращает ссылку `this`, т.е. текущую метку. (Напомним, что класс `DefaultTreeCellRenderer` расширяет класс `JLabel`.) Чтобы создать специальный компонент, предназначенный для воспроизведения ячеек дерева, необходимо расширить сначала класс `DefaultTreeCellRenderer`, а затем переопределить метод `getTreeCellRendererComponent()`, предусмотрев в нем вызов из суперкласса того метода, который подготовит все данные, необходимые для создания метки. Кроме того, в данном методе задаются требующиеся значения свойств, а по завершении своего выполнения он возвращает ссылку `this`.

```
class MyTreeCellRenderer extends DefaultTreeCellRenderer
{
  public Component getTreeCellRendererComponent(JTree tree,
         Object value, boolean selected, boolean expanded,
         boolean leaf, int row, boolean hasFocus)
  {
     Component comp = super.getTreeCellRendererComponent(
                          tree, value, selected,
                          expanded, leaf, row, hasFocus);
     DefaultMutableTreeNode node =
                  (DefaultMutableTreeNode) value;
     извлечь пользовательский объект из данного узла:
     node.getUserObject();
     Font font = подходящий шрифт;
     comp.setFont(font);
     return comp;
  }
};
```

ВНИМАНИЕ! Параметр **value** метода **getTreeCellRendererComponent()** является *узловым*, а *не* пользовательским объектом. Напомним, что пользовательский объект относится к классу узлов **DefaultMutableTreeNode**, а класс **JTree** может содержать узлы произвольного типа. Если в дереве используются узлы типа **DefaultMutableTreeNode**, то на второй стадии следует извлечь пользовательский объект, как это было сделано в предыдущем примере кода.

ВНИМАНИЕ! В классе **DefaultTreeCellRenderer** используется один и тот же объект метки для всех узлов, но для каждого узла изменяется ее текст. Если изменить шрифт для выделения названия отдельного узла, то при последующем вызове упомянутого выше метода следует восстановить используемый по умолчанию шрифт. В противном случае названия всех последующих узлов будут выделены новым шрифтом! Один из способов восстановления исходного шрифта представлен далее в листинге 11.10.

Средство воспроизведения типа `ClassNameTreeCellRenderer` из листинга 11.10 выделяет имя класса обычным шрифтом или курсивом в зависимости от наличия модификатора `ABSTRACT` у объекта типа `Class`. Для обозначения абстрактных классов не выбирается какой-то другой шрифт, чтобы не изменять визуальный стиль, обычно применяемый для отображения дерева. По этой причине курсив для обозначения абстрактного класса получается путем наклона начертания исходного шрифта, которым выделяются метки. Напомним, что в результате всех вызовов возвращается только один общий объект типа `JLabel`. При последующих вызовах метода `getTreeCellRendererComponent()` необходимо снова вернуться к исходному шрифту. Обратите также внимание, каким образом изменяются пиктограммы узлов в конструкторе класса `ClassTreeFrame`.

javax.swing.tree.DefaultMutableTreeNode 1.2

- **Enumeration breadthFirstEnumeration()**
- **Enumeration depthFirstEnumeration()**
- **Enumeration preOrderEnumeration()**
- **Enumeration postOrderEnumeration()**

  Возвращают объект типа **Enumeration**, представляющий результаты обхода всех узлов в модели дерева. При обходе в ширину дочерние узлы, находящиеся ближе к корневому узлу, посещаются раньше. При обходе в глубину перед переходом к равноправному узлу посещаются все дочерние узлы. Метод **postOrderEnumeration()** является аналогом метода **depthFirstEnumeration()**. Обход в ширину (или в прямом порядке) подобен обходу в глубину (или в обратном порядке) за исключением того, что родительские узлы перечисляются прежде дочерних.

javax.swing.tree.TreeCellRenderer 1.2

- **Component getTreeCellRendererComponent(JTree *tree*, Object *value*, boolean *selected*, boolean *expanded*, boolean *leaf*, int *row*, boolean *hasFocus*)**

  Возвращает компонент, метод **paint()** которого вызывается для воспроизведения ячейки (или узла) дерева.

Параметры:	***tree***	Дерево, содержащее воспроизводимый узел
	value	Воспроизводимый узел
	selected	Принимает логическое значение **true**, если данный узел выбран

javax.swing.tree.TreeCellRenderer 1.2 *(окончание)*

expanded	Принимает логическое значение **true**, если видны дочерние узлы данного узла дерева
leaf	Принимает логическое значение **true**, если данный узел должен отображаться как лист
row	Отображаемая строка, содержащая узел дерева
hasFocus	Принимает логическое значение **true**, если данный узел обладает фокусом ввода

javax.swing.tree.DefaultTreeCellRenderer 1.2

- **void setLeafIcon(Icon *icon*)**
- **void setOpenIcon(Icon *icon*)**
- **void setClosedIcon(Icon *icon*)**

  Задают пиктограмму для обозначения листового, развернутого или свернутого узла.

11.2.4. Обработка событий в деревьях

Дерево чаще всего используется совместно с каким-нибудь другим компонентом. Например, при выборе какого-нибудь узла дерева в соседнем окне может быть показана определенная сопроводительная информация. На рис. 11.29 представлен пример программы, где для каждого узла в правой текстовой области автоматически отображаются статические переменные и переменные экземпляра соответствующего класса.

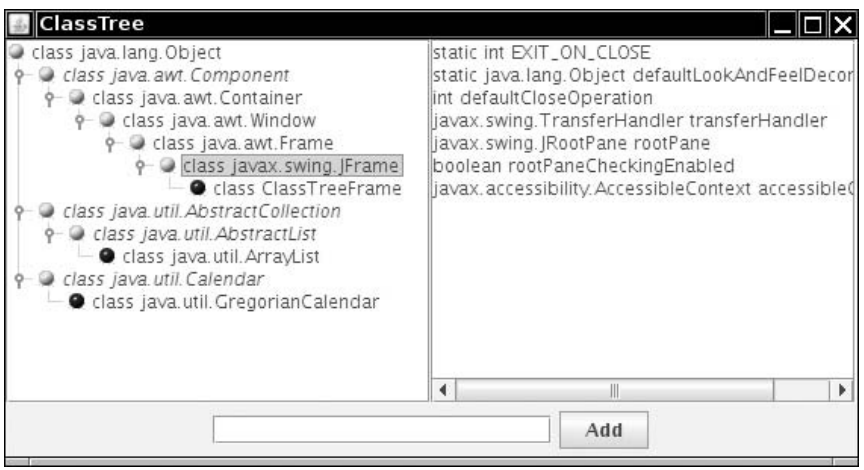


Рис. 11.29. Пример программы для просмотра классов

Чтобы добиться такого поведения, необходимо установить *приемник событий выбора из дерева*. Класс такого приемника событий должен реализовать

интерфейс `TreeSelectionListener` со следующим единственным методом `valueChanged()`:

```
void valueChanged(TreeSelectionEvent event)
```

Метод `valueChanged()` вызывается при каждом событии выбора или отмены выбора узлов дерева. Приемник событий добавляется к дереву обычным образом:

```
tree.addTreeSelectionListener(listener);
```

С помощью свойств модели типа `TreeSelectionModel` в классе `JTree` допускается устанавливать конкретный режим выбора узлов дерева: только один узел (свойство `SINGLE_TREE_SELECTION`), группа смежных узлов (свойство `CONTINGUOUS_TREE_SELECTION`) или произвольная группа несмежных узлов (свойство `DISCONTINGUOUS_TREE_SELECTION`). По умолчанию допускается выбор произвольной группы несмежных узлов. В рассматриваемом здесь примере программы для просмотра классов допускается выбор только одного узла дерева, как показано ниже. Никаких других действий, кроме установки конкретного режима выбора, предпринимать не нужно.

```
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
tree.getSelectionModel().setSelectionMode(mode);
```

НА ЗАМЕТКУ! Порядок выбора нескольких элементов зависит от конкретного визуального стиля пользовательского интерфейса. Так, если применяется визуальный стиль Metal, для выбора несмежных элементов (в данном случае — узлов дерева) следует нажать клавишу <Ctrl> и, не отпуская ее, щелкнуть по очереди на каждом выбираемом элементе, а для выбора нескольких смежных элементов — нажать клавишу <Shift> и, не отпуская ее, щелкнуть сначала на первом, а затем на последнем элементе из выбираемой группы.

Чтобы выяснить, что именно выбрано в настоящий момент, необходимо запросить дерево, вызвав метод `getSelectionPaths()` следующим образом:

```
TreePath[] selectedPaths = tree.getSelectionPaths();
```

Если же требуется ограничить возможности пользователя, разрешив ему выбирать узлы дерева только по одному, то удобнее воспользоваться служебным методом `getSelectionPath()`, который в общем случае возвращает первый же выбранный путь к элементу или пустое значение `null`, если такой путь не выбран.

ВНИМАНИЕ! В классе **`TreeSelectionEvent`** имеется метод **`getPaths()`**, который возвращает массив объектов типа **`TreePath`**, но этот массив описывает *изменения* в самом выборе, а *не* текущий результат выбора.

В листинге 11.10 приведен исходный код класса фрейма для программы, отображающей иерархию наследования классов в виде древовидной структуры, где абстрактные классы выделены курсивом. В листинге 11.11 представлен исходный код, реализующий средство воспроизведения ячеек дерева. Если ввести полное имя класса и нажать клавишу <Enter> или щелкнуть на кнопке Add, в дерево будет введен новый класс со всеми его суперклассами. В полное имя класса следует включить имя пакета, например `java.util.ArrayList`.

Рассматриваемая здесь программа немного трудна для понимания, поскольку в ней для построения дерева классов используется рефлексия. Весь код построения дерева классов находится в методе `addClass()`. (Иерархия наследования классов используется здесь лишь в качестве удобного примера для демонстрации особенностей построения деревьев и обращения с ними, не усложняя программирование. При разработке собственных прикладных программ можно использовать любые другие источники иерархических данных.) Для проверки наличия вводимого класса в дереве вызывается метод `findUserObject()`, реализация которого описана в предыдущем разделе. В этом методе реализуется алгоритм обхода дерева в ширину. Если класс отсутствует в дереве, то сначала вводятся его суперклассы, а затем создается и отображается новый узел для данного класса.

Когда выбран узел дерева, текстовая область справа заполняется полями выбранного класса. В конструкторе фрейма накладываются ограничения, разрешающие пользователю данной программы выбирать узлы дерева только по одному, а также вводится приемник событий выбора из дерева. При вызове метода `valueChanged()` его параметр события игнорируется, а у дерева запрашивается только путь к текущему выбранному узлу. Далее последний узел, как обычно, получается по заданному пути, и из него извлекается пользовательский объект. После этого вызывается метод `getFieldDescription()`, где для формирования символьной строки со всеми полями выбранного класса применяется рефлексия.

Листинг 11.10. Исходный код из файла **`treeRender/ClassTreeFrame.java`**

```
1   package treeRender;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.lang.reflect.*;
6   import java.util.*;
7
8   import javax.swing.*;
9   import javax.swing.tree.*;
10
11  /**
12   * В этом фрейме отображается дерево иерархии классов,
13   * текстовое поле и экранная кнопка Add для ввода
14   * новых классов в дерево
15   */
16  public class ClassTreeFrame extends JFrame
17  {
18    private static final int DEFAULT_WIDTH = 400;
19    private static final int DEFAULT_HEIGHT = 300;
20
21    private DefaultMutableTreeNode root;
22    private DefaultTreeModel model;
23    private JTree tree;
24    private JTextField textField;
25    private JTextArea textArea;
26
27    public ClassTreeFrame()
28    {
```

```
29      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
30
31      // в корне дерева иерархии классов
32      // находится класс Object:
33      root = new DefaultMutableTreeNode(
34                java.lang.Object.class);
35      model = new DefaultTreeModel(root);
36      tree = new JTree(model);
37
38      // ввести этот класс, чтобы заполнить
39      // дерево некоторыми данными:
40      addClass(getClass());
41
42      // установить пиктограммы для обозначения узлов:
43      var renderer = new ClassNameTreeCellRenderer();
44      renderer.setClosedIcon(new ImageIcon(getClass()
45                      .getResource("red-ball.gif")));
46      renderer.setOpenIcon(new ImageIcon(getClass()
47                      .getResource("yellow-ball.gif")));
48      renderer.setLeafIcon(new ImageIcon(getClass()
49                      .getResource("blue-ball.gif")));
50      tree.setCellRenderer(renderer);
51
52      // установить режим выбора узлов дерева:
53      tree.addTreeSelectionListener(event ->
54        {
55          // пользователь выбрал другой узел —
56          // обновить его описание:
57          TreePath path = tree.getSelectionPath();
58          if (path == null) return;
59          var selectedNode = (DefaultMutableTreeNode)
60                           path.getLastPathComponent();
61          Class<?> c = (Class<?>)
62                        selectedNode.getUserObject();
63          String description = getFieldDescription(c);
64          textArea.setText(description);
65        });
66      int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
67      tree.getSelectionModel().setSelectionMode(mode);
68
69      // в этой текстовой области находится
70      // описание класса:
71      textArea = new JTextArea();
72
73      // добавить дерево и текстовую область:
74      var panel = new JPanel();
75      panel.setLayout(new GridLayout(1, 2));
76      panel.add(new JScrollPane(tree));
77      panel.add(new JScrollPane(textArea));
78
79      add(panel, BorderLayout.CENTER);
80
81      addTextField();
82    }
83
84    /**
```

```
85     * Добавляет в фрейм текстовое поле и экранную
86     * кнопку Add для ввода нового класса
87     */
88    public void addTextField()
89    {
90      var panel = new JPanel();
91
92      ActionListener addListener = event ->
93        {
94          // ввести класс, имя которого находится
95          // в текстовом поле:
96          try
97          {
98            String text = textField.getText();
99            // очистить текстовое поле, чтобы
100           // обозначить удачный исход ввода класса:
101           addClass(Class.forName(text));
102           textField.setText("");
103         }
104         catch (ClassNotFoundException e)
105         {
106           JOptionPane.showMessageDialog(
107                       null, "Class not found");
108         }
109       };
110
111     // в этом текстовом поле вводятся
112     // имена новых классов:
113     textField = new JTextField(20);
114     textField.addActionListener(addListener);
115     panel.add(textField);
116
117     var addButton = new JButton("Add");
118     addButton.addActionListener(addListener);
119     panel.add(addButton);
120
121     add(panel, BorderLayout.SOUTH);
122   }
123
124   /**
125    * Находит искомый объект в дереве
126    * @param obj Искомый объект
127    * @return Узел с объектом или пустое значение null,
128    *         если объект отсутствует в дереве
129    */
130   public DefaultMutableTreeNode
131         findUserObject(Object obj)
132   {
133     // найти узел, содержащий пользовательский объект:
134     var e = (Enumeration<TreeNode>)
135             root.breadthFirstEnumeration();
136     while (e.hasMoreElements())
137     {
138       var node = (DefaultMutableTreeNode)
139                   e.nextElement();
140       if (node.getUserObject().equals(obj))
```

```
141         return node;
142       }
143       return null;
144     }
145
146     /**
147      * Вводит новый класс и любые его родительские
148      * классы, пока еще отсутствующие в дереве
149      * @param c Вводимый класс
150      * @return Узел с вновь введенным классом
151      */
152     public DefaultMutableTreeNode addClass(Class<?> c)
153     {
154       // ввести новый класс в дерево
155
156       // пропустить типы данных, не относящиеся к классам:
157       if (c.isInterface() || c.isPrimitive())
158         return null;
159
160       // если класс уже присутствует в дереве,
161       // возвратить его узел:
162       DefaultMutableTreeNode node = findUserObject(c);
163       if (node != null) return node;
164
165       // класс отсутствует в дереве — ввести сначала
166       // его родительские классы рекурсивным способом
167
168       Class<?> s = c.getSuperclass();
169
170       DefaultMutableTreeNode parent;
171       if (s == null) parent = root;
172       else parent = addClass(s);
173
174       // ввести затем класс как потомок
175       // его родительского класса:
176       var newNode = new DefaultMutableTreeNode(c);
177       model.insertNodeInto(newNode, parent,
178                            parent.getChildCount());
179
180       // сделать видимым узел с вновь введенным классом:
181       var path = new TreePath(
182                  model.getPathToRoot(newNode));
183       tree.makeVisible(path);
184
185       return newNode;
186     }
187
188     /**
189      * Возвращает описание полей класса
190      * @param Описываемый класс
191      * @return Символьная строка, содержащая все
192      *         типы и имена полей описываемого класса
193      */
194     public static String getFieldDescription(Class<?> c)
195     {
196       // использовать рефлексию для обнаружения
```

```
197     // типов и имен полей:
198     var r = new StringBuilder();
199     Field[] fields = c.getDeclaredFields();
200     for (int i = 0; i < fields.length; i++)
201     {
202       Field f = fields[i];
203       if ((f.getModifiers() & Modifier.STATIC) != 0)
204         r.append("static ");
205       r.append(f.getType().getName());
206       r.append(" ");
207       r.append(f.getName());
208       r.append("\n");
209     }
210     return r.toString();
211   }
212 }
```

Листинг 11.11. Исходный код из файла **treeRender/ClassNameTreeCellRenderer.java**

```
1  package treeRender;
2
3  import java.awt.*;
4  import java.lang.reflect.*;
5  import javax.swing.*;
6  import javax.swing.tree.*;
7  /**
8   * Этот класс воспроизводит имя класса, выделяя его
9   * простым шрифтом или курсивом. Абстрактные классы
10  * выделяются только курсивом
11  */
12 public class ClassNameTreeCellRenderer
13        extends DefaultTreeCellRenderer
14 {
15   private Font plainFont = null;
16   private Font italicFont = null;
17
18   public Component getTreeCellRendererComponent(
19          JTree tree, Object value, boolean selected,
20          boolean expanded, boolean leaf, int row,
21          boolean hasFocus)
22   {
23     super.getTreeCellRendererComponent(tree, value,
24           selected, expanded, leaf, row, hasFocus);
25     // получить пользовательский объект:
26     var node = (DefaultMutableTreeNode) value;
27     Class<?> c = (Class<?>) node.getUserObject();
28
29     // сначала сделать простой шрифт наклонным:
30     if (plainFont == null)
31     {
32       plainFont = getFont();
33       // средство воспроизведения ячеек дерева иногда
34       // вызывается с меткой, имеющей пустой шрифт
```

```
35        if (plainFont != null)
36          italicFont = plainFont.deriveFont(Font.ITALIC);
37      }
38
39      // установить наклонный шрифт, если класс является
40      // абстрактным, а иначе — простой шрифт:
41      if ((c.getModifiers() & Modifier.ABSTRACT) == 0)
42        setFont(plainFont);
43      else
44        setFont(italicFont);
45      return this;
46    }
47 }
```

javax.swing.JTree 1.2

- **TreePath getSelectionPath()**
- **TreePath[] getSelectionPaths()**

  Возвращают путь к первому выбранному узлу или массив путей ко всем выбранным узлам дерева. Если ни один из узлов дерева не выбран, оба метода возвращают пустое значение **null**.

javax.swing.event.TreeSelectionListener 1.2

- **void valueChanged(TreeSelectionEvent *event*)**

  Вызывается всякий раз, когда происходит выбор или отмена выбора узлов дерева.

javax.swing.event.TreeSelectionEvent 1.2

- **TreePath getPath()**
- **TreePath[] getPaths()**

  Получают первый путь или все пути, которые *изменились* в результате данного события выбора из дерева. Для получения сведений о текущем выборе, а не изменении выбора следует вызывать метод **JTree.getSelectionPath()**.

11.2.5. Специальные модели деревьев

В качестве последнего примера манипулирования деревьями рассмотрим программу, которая, подобно отладчику, проверяет содержимое переменной или объекта. На рис. 11.30 показано рабочее окно данной программы.

Скомпилируйте и запустите эту программу на выполнение, прежде чем читать ее описание. Каждый узел дерева соответствует какому-нибудь полю экземпляра. Если поле содержит объект, разверните структуру этого объекта для просмотра полей экземпляра его класса. Как видите, данная программа позволяет просматривать содержимое своего фрейма. При более внимательном изучении

нескольких полей экземпляра вы можете обнаружить некоторые уже знакомые вам классы. Кроме того, эта программа позволяет оценить, насколько сложны компоненты библиотеки Swing, предназначенные для построения пользовательского интерфейса.

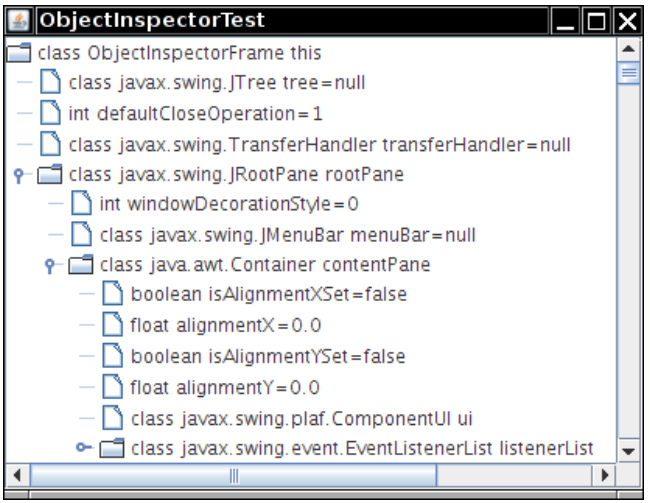


Рис. 11.30. Дерево для обследования объектов

Данная программа примечательна тем, что для построения дерева в ней не применяется модель типа `DefaultTreeModel`. При наличии данных, которые уже иерархически организованы в виде древовидной структуры, вряд ли имеет смысл использовать предлагаемое по умолчанию дерево и синхронизировать оба дерева. Именно такая ситуация возникает в рассматриваемом здесь примере: обследуемые объекты уже связаны друг с другом с помощью ссылок на них, и поэтому нет никакой нужды дублировать структуру их связей.

В интерфейсе `TreeModel` объявлено лишь несколько удобных методов. Первая группа методов позволяет найти узлы дерева, начиная с корня и продолжая дочерними узлами. Эти методы вызываются в классе `JTree` только в том случае, если узел был развернут пользователем:

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

В рассматриваемом здесь примере наглядно демонстрируется, почему для интерфейса `TreeModel`, как и для класса `JTree`, не требуется явное обозначение узлов. Корневые и дочерние узлы могут быть объектами произвольного типа, интерфейс `TreeModel` отвечает за уведомление класса `JTree` об установлении связи между ними. Следующий метод из интерфейса `TreeModel` выполняет действия, обратные методу `getChild()`:

```
int getIndexOfChild(Object parent, Object child)
```

Как следует из листинга 11.12, этот метод можно реализовать на основе первых трех методов. Модель дерева сообщает компоненту `JTree`, какие именно узлы следует отобразить как листовые:

```
boolean isLeaf(Object node)
```

Если при выполнении кода изменяется модель дерева, то дерево следует уведомить о необходимости его воспроизведения заново. Поэтому дерево вводится в модель как приемник событий типа `TreeModelListener`. Следовательно, в модели должны использоваться обычные методы управления обработчиками событий, как показано ниже, а их реализация представлена в листинге 11.13.

```
void addTreeModelListener(TreeModelListener l)
void removeTreeModelListener(TreeModelListener l)
```

Для изменения содержимого дерева в его модели вызывается один из четырех перечисленных ниже методов из интерфейса `TreeModelListener`.

```
void treeNodesChanged(TreeModelEvent e)
void treeNodesInserted(TreeModelEvent e)
void treeNodesRemoved(TreeModelEvent e)
void treeStructureChanged(TreeModelEvent e)
```

Объект типа `TreeModelEvent` описывает место, где происходят изменения в дереве. Подробности организации событий, наступающих в модели дерева при вводе и удалении узлов, здесь не рассматриваются в силу того, что они носят слишком технический характер. Вам нужно лишь позаботиться об инициировании подобных событий, когда в дереве вводятся и удаляются отдельные узлы. В листинге 11.12 демонстрируется пример инициирования события, наступающего при замене корневого узла новым объектом.

СОВЕТ. Для упрощения кода инициирования событий рекомендуется удобный класс **`javax.swing.EventListenerList`**, предназначенный для составления списка из приемников событий. На примере трех последних методов из листинга 11.13 показано, как пользоваться этим классом в прикладном коде.

Наконец, если пользователь редактирует узел дерева, то его модель вызывается с помощью следующего метода, где указывается вносимое изменение:

```
void valueForPathChanged(TreePath path, Object newValue)
```

Если же редактирование узлов не разрешается, этот метод вообще не вызывается. В таком случае построить модель еще проще. Для этого достаточно реализовать три метода:

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

Эти методы описывают структуру дерева. После реализации остальных пяти методов, как показано в листинге 11.12, можно приступать к отображению дерева.

Теперь перейдем непосредственно к реализации рассматриваемого здесь примера программы. В ней строится дерево, состоящее из объектов типа `Variable` в его узлах.

НА ЗАМЕТКУ! Если бы в данном примере применялась модель дерева типа **DefaultTreeModel**, то узлы были бы объектами типа **DefaultMutableTreeNode** с *пользовательскими объектами* типа **Variable**.

Допустим, требуется проверить переменную, объявленную следующим образом:

```
Employee joe;
```

Эта переменная имеет *тип* `Employee.class`, *имя* `joe` и *значение* в виде ссылки на объект `joe`. В листинге 11.14 объект класса `Variable`, описывающего эту переменную в рассматриваемом здесь примере программы, определяется следующим образом:

```
var v = new Variable(Employee.class, "joe", joe);
```

Если переменная относится к примитивному типу, то для ее значения нужно создать объектную оболочку:

```
new Variable(double.class, "salary", new Double(salary));
```

Если же переменная относится к типу класса, то она имеет *поля*, которые можно перечислить и собрать в объекте типа `ArrayList` с помощью рефлексии. Метод `getFields()` из класса `Class` не возвращает поля суперкласса, поэтому данный метод придется вызвать для всех суперклассов проверяемого класса. Для этой цели служит исходный код, находящийся в конструкторе класса `Variable`. Метод `getFields()` из класса `Variable` возвращает массив полей. Наконец, метод `toString()` из класса `Variable` форматирует метку узла дерева. Метка всегда содержит тип и имя переменной, а если переменная не относится к типу класса, то метка содержит также ее значение.

НА ЗАМЕТКУ! Если тип переменной представлен массивом, то его элементы в данном примере не отображаются. Впрочем, сделать это совсем не трудно, и такая возможность предоставляется читателям в качестве дополнительного упражнения.

Перейдем непосредственно к модели дерева. Исходный код первых двух ее методов несложен:

```
public Object getRoot()
{
  return root;
}

public int getChildCount(Object parent)
{
  return ((Variable) parent).getFields().size();
}
```

Метод `getChild()` возвращает новый объект типа `Variable`, описывающий поле по указанному индексу. С помощью методов `getType()` и `getName()` из класса `Field` можно получить тип и имя поля, а с помощью рефлексии — прочитать значение поля, вызвав метод `f.get(parentValue)`. Этот метод может сгенерировать исключение типа `IllegalAccessException`. Но в конструкторе класса `Variable` предоставлен свободный доступ ко всем полям, поэтому подобная

исключительная ситуация вряд ли вообще возникнет. Ниже приведен весь исходный код метода `getChild()`.

```
public Object getChild(Object parent, int index)
{
  ArrayList fields = ((Variable) parent).getFields();
  var f = (Field) fields.get(index);
  Object parentValue = ((Variable) parent).getValue();
  try
  {
    return new Variable(f.getType(), f.getName(), f.get(parentValue));
  }
  catch (IllegalAccessException e)
  {
    return null;
  }
}
```

Упомянутые выше три метода выявляют структуру дерева объектов для компонента `JTree`. Остальные методы выполняют другие рутинные операции, как следует из листинга 11.13.

Необходимо отметить следующую особенность рассматриваемой здесь модели дерева: она описывает *бесконечное* дерево. Это можно проверить на примере одного из объектов типа `WeakReference`. Так, если щелкнуть в дереве на имени переменной `referent`, произойдет возврат к исходному объекту, который содержит идентичное подчиненное дерево с объектом типа `WeakReference`. Разумеется, сохранить бесконечное количество узлов невозможно, поэтому данная модель дерева формирует дочерние узлы по требованию в процессе постепенного развертывания родительских узлов. Исходный код класса фрейма для данной программы представлен в листинге 11.12.

Листинг 11.12. Исходный код из файла **`treeModel/ObjectInspectorFrame.java`**

```
1  package treeModel;
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  /**
7   * Этот фрейм содержит дерево объектов
8   */
9  public class ObjectInspectorFrame extends JFrame
10 {
11   private JTree tree;
12   private static final int DEFAULT_WIDTH = 400;
13   private static final int DEFAULT_HEIGHT = 300;
14
15   public ObjectInspectorFrame()
16   {
17     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
18
19     // обследовать объект данного фрейма:
20
```

```
21      var v = new Variable(getClass(), "this", this);
22      var model = new ObjectTreeModel();
23      model.setRoot(v);
24
25      // построить и показать дерево:
26
27      tree = new JTree(model);
28      add(new JScrollPane(tree), BorderLayout.CENTER);
29    }
30 }
```

Листинг 11.13. Исходный код из файла **treeModel/ObjectTreeModel.java**

```
1   package treeModel;
2
3   import java.lang.reflect.*;
4   import java.util.*;
5   import javax.swing.event.*;
6   import javax.swing.tree.*;
7
8   /**
9    * Эта модель дерева описывает древовидную структуру
10   * объектов в Java. Дочерние узлы дерева являются
11   * объектами, хранящимися в переменных экземпляра
12  */
13  public class ObjectTreeModel implements TreeModel
14  {
15    private Variable root;
16    private EventListenerList listenerList =
17            new EventListenerList();
18
19    /**
20     * Строит пустое дерево
21     */
22    public ObjectTreeModel()
23    {
24      root = null;
25    }
26
27    /**
28     * Устанавливает заданную переменную в корне дерева
29     * @param v Переменная, описываемая в данном дереве
30    */
31    public void setRoot(Variable v)
32    {
33      Variable oldRoot = v;
34      root = v;
35      fireTreeStructureChanged(oldRoot);
36    }
37
38    public Object getRoot()
39    {
40      return root;
41    }
42
```

```
43    public int getChildCount(Object parent)
44    {
45      return ((Variable) parent).getFields().size();
46    }
47
48    public Object getChild(Object parent, int index)
49    {
50      ArrayList<Field> fields =
51                      ((Variable) parent).getFields();
52      var f = (Field) fields.get(index);
53      Object parentValue = ((Variable) parent).getValue();
54      try
55      {
56        return new Variable(f.getType(), f.getName(),
57                            f.get(parentValue));
58      }
59      catch (IllegalAccessException e)
60      {
61        return null;
62      }
63    }
64
65    public int getIndexOfChild(Object parent,
66                               Object child)
67    {
68      int n = getChildCount(parent);
69      for (int i = 0; i < n; i++)
70        if (getChild(parent, i).equals(child))
71          return i;
72      return -1;
73    }
74
75    public boolean isLeaf(Object node)
76    {
77      return getChildCount(node) == 0;
78    }
79
80    public void valueForPathChanged(
81                TreePath path, Object newValue)
82    {
83    }
84
85    public void addTreeModelListener(TreeModelListener l)
86    {
87      listenerList.add(TreeModelListener.class, l);
88    }
89
90    public void removeTreeModelListener(
91                TreeModelListener l)
92    {
93      listenerList.remove(TreeModelListener.class, l);
94    }
95
96    protected void fireTreeStructureChanged(
97                   Object oldRoot)
98    {
```

```
99      var event = new TreeModelEvent(this,
100                 new Object[] { oldRoot });
101     for (TreeModelListener l : listenerList
102             .getListeners(TreeModelListener.class))
103       l.treeStructureChanged(event);
105   }
106 }
```

Листинг 11.14. Исходный код из файла **treeModel/Variable.java**

```
1  package treeModel;
2
3  import java.lang.reflect.*;
4  import java.util.*;
5
6  /**
7   * Переменная с типом, именем и значением
8  */
9  public class Variable
10 {
11   private Class<?> type;
12   private String name;
13   private Object value;
14   private ArrayList<Field> fields;
15
16   /**
17    * Сконструировать переменную
18    * @param aType Тип
19    * @param aName Имя
20    * @param aValue Значение
21   */
22   public Variable(Class<?> aType, String aName,
23                   Object aValue)
24   {
25     type = aType;
26     name = aName;
27     value = aValue;
28     fields = new ArrayList<>();
29
30     // найти все поля, если это тип класса,
31     // за исключением символьных строк и
32     // пустых значений null
33
34     if (!type.isPrimitive() && !type.isArray()
35         && !type.equals(String.class) && value != null)
36     {
37       // получить поля из класса и всех его суперклассов:
38       for (Class<?> c = value.getClass(); c != null;
39            c = c.getSuperclass())
40       {
41         Field[] fs = c.getDeclaredFields();
42         AccessibleObject.setAccessible(fs, true);
43
44         // получить все нестатические поля:
45         for (Field f : fs)
```

```
46            if ((f.getModifiers() & Modifier.STATIC) == 0)
47               fields.add(f);
48         }
49       }
50    }
51
52    /**
53     * Получает значение данной переменной
54     * @return Возвращает значение
55    */
56    public Object getValue()
57    {
58      return value;
59    }
60
61    /**
62     * Получает все нестатические поля из данной переменной
63     * @return Списочный массив переменных
64     *         с описаниями полей
65     */
66    public ArrayList<Field> getFields()
67    {
68      return fields;
69    }
70
71    public String toString()
72    {
73      String r = type + " " + name;
74      if (type.isPrimitive()) r += "=" + value;
75      else if (type.equals(String.class)) r += "=" + value;
76      else if (value == null) r += "=null";
77      return r;
78    }
79 }
```

javax.swing.tree.TreeModel 1.2

- **Object getRoot()**

  Возвращает корневой узел дерева.

- **int getChildCount(Object *parent*)**

  Получает количество дочерних узлов заданного родительского узла дерева.

- **Object getChild(Object *parent*, int *index*)**

  Получает дочерний узел заданного родительского узла дерева по указанному индексу.

- **int getIndexOfChild(Object *parent*, Object *child*)**

  Получает индекс указанного дочернего узла из заданного родительского узла дерева или значение **-1**, если узел ***child*** не является дочерним родительского узла ***parent*** в данной модели дерева.

- **boolean isLeaf(Object *node*)**

  Возвращает логическое значение **true**, если указанный узел является листом дерева.

javax.swing.tree.TreeModel 1.2 *(окончание)*

- **void addTreeModelListener(TreeModelListener *l*)**
- **void removeTreeModelListener(TreeModelListener *l*)**

  Вводят или удаляют приемник событий, который уведомляется о происходящих в дереве изменениях.
- **void valueForPathChanged(TreePath *path*, Object *newValue*)**

  Вызывается в тех случаях, когда значение узла изменяется в редакторе ячеек дерева.

  Параметры: ***path*** Путь к отредактированному узлу

  newValue Замещающее значение, возвращаемое редактором ячеек дерева

javax.swing.event.TreeModelListener 1.2

- **void treeNodesChanged(TreeModelEvent *e*)**
- **void treeNodesInserted(TreeModelEvent *e*)**
- **void treeNodesRemoved(TreeModelEvent *e*)**
- **void treeStructureChanged(TreeModelEvent *e*)**

  Вызываются моделью при изменении узлов дерева.

javax.swing.event.TreeModelEvent 1.2

- **TreeModelEvent(Object *eventSource*, TreePath *node*)**

  Конструирует событие в модели дерева.

  Параметры: ***eventSource*** Модель дерева, инициирующая данное событие

  node Путь к изменяемому узлу

11.3. Расширенные средства AWT

Для создания простых рисунков можно использовать методы из класса `Graphics`. Они эффективны для разработки простых приложений, но их недостаточно для построения сложных фигур или полного контроля над внешним видом рисунков. В состав прикладного интерфейса Java 2D API входит библиотека классов для создания высококачественных рисунков. В последующих разделах дается краткий обзор этого прикладного интерфейса.

11.3.1. Конвейер визуализации

В исходной версии JDK 1.0 применялся очень простой механизм для рисования фигур. Для этого достаточно было выбрать нужный цвет, режим рисования и вызвать подходящие методы из класса `Graphics`, например `drawRect()` или

`fillOval()`. В прикладном интерфейсе Java 2D API поддерживается намного больше возможностей для рисования двухмерной графики. Он, в частности, позволяет делать следующее.

- Легко формировать самые разные фигуры.
- Управлять *обводкой*, т.е. вычерчивать пером контуры фигур.
- Заполнять фигуры любым сплошным цветом, используя различные оттенки и узоры.
- Выполнять *преобразования* для перемещения, масштабирования, вращения и растягивания фигур.
- *Отсекать* фигуры таким образом, чтобы ограничить их произвольно выбираемым участком экрана.
- Выбирать *правила композиции*, чтобы описывать порядок сочетания пикселей новой и уже существующей фигур.
- Давать *указания по воспроизведению* для достижения компромисса между скоростью загрузки и качеством рисования.

Чтобы нарисовать фигуру, необходимо выполнить следующие действия.

1. Получить объект класса `Graphics2D`. Это подкласс, производный от класса `Graphics`. Начиная с версии Java 1.2 такие методы, как `paint()` и `paintComponent()`, автоматически получают объект класса `Graphics2D`. Поэтому остается только выполнить приведение типов, как показано ниже.

```
public void paintComponent(Graphics g)
{
  var g2 = (Graphics2D) g;
  . . .
}
```

2. Вызвать приведенный ниже метод `setRenderingHints()`, чтобы добавить указания по воспроизведению для достижения компромисса между скоростью и качеством рисования.

```
RenderingHints hints = . . .;
g2.setRenderingHints(hints);
```

3. Вызвать приведенный ниже метод `setStroke()`, чтобы задать *обводку* контура фигуры. Для обводки можно выбрать толщину, а также сплошную или пунктирную линию.

```
Stroke stroke = . . .;
g2.setStroke(stroke);
```

4. Вызвать приведенный ниже метод `setPaint()`, чтобы указать способ *раскраски*. Раскраска подразумевает закрашивание участков контура обводки или внутренней области фигуры. Она может состоять из одного сплошного цвета, нескольких меняющихся оттенков или мозаичных узоров.

```
Paint paint = . . .;
g2.setPaint(paint);
```

5. Вызвать метод `clip()`, чтобы задать *область отсечения* следующим образом:

```
Shape clip = . . .;
g2.clip(clip);
```

6. Вызвать приведенный ниже метод `setTransform()` для *преобразования* рисунка из пользовательского пространства в пространство конкретного устройства. Такое преобразование следует выполнять в тех случаях, когда фигуру проще создать в специальной системе координат, чем использовать для этой цели координаты, выражаемые в пикселях.

```
AffineTransform transform = . . .;
g2.transform(transform);
```

7. Вызвать приведенный ниже метод `setComposite()`, чтобы задать правило композиции, описывающее порядок объединения новых пикселей с уже существующими пикселями.

```
Composite composite = . . .;
g2.setComposite(composite);
```

8. Создать фигуру, как показано ниже. В прикладном интерфейсе Java 2D API предусмотрено немало объектов фигур и методов для их сочетания.

```
Shape shape = . . .;
```

9. Нарисовать или заполнить фигуру, как показано ниже. Под рисованием подразумевается очерчивание контуров фигуры, а под заполнением — закрашивание ее внутренней области.

```
g2.draw(shape);
g2.fill(shape);
```

Разумеется, на практике выполнять все эти действия обычно не требуется, поэтому для многих параметров двухмерного графического контекста предусмотрены значения по умолчанию, изменять которые следует только в случае особой необходимости. В последующих разделах рассматриваются способы описания фигур, обводок, раскрасок, преобразований и правил композиции.

Различные методы типа `set` просто устанавливают состояние двухмерного графического контекста и не относятся к конкретным рисункам. Аналогично фигура воспроизводится не во время создания объекта типа `Shape`, а при вызове методов `draw()` или `fill()`, когда новая фигура рассчитывается в *конвейере визуализации* (рис. 11.31).

Рис. 11.31. Конвейер визуализации

Для воспроизведения фигуры в конвейере визуализации выполняются следующие действия.

1. Обводится контур фигуры.
2. Выполняется преобразование фигуры.
3. Происходит отсечение фигуры. Данный процесс прекращается, как только фигура перестает пересекать область отсечения.
4. Оставшаяся после отсечения часть фигуры заполняется.
5. Составляется в единый рисунок композиция из пикселей фигуры и уже существующих пикселей. (Как показано на рис. 11.31, существующие пиксели образуют круг, а фигура чашки накладывается на него.)

В следующем разделе сначала рассматриваются способы определения фигур, а затем порядок установки двухмерного графического контекста.

java.awt.Graphics2D 1.2

- **void draw(Shape *s*)**
  Рисует контур указанной фигуры в соответствии с текущей раскраской.
- **void fill(Shape *s*)**
  Заполняет внутреннюю область фигуры в соответствии с текущей раскраской.

11.3.2. Фигуры

Ниже перечислен ряд методов из класса `Graphics`, применяемых для рисования фигур.

```
DrawLine()
drawRectangle()
drawRoundRect()
draw3Drect()
drawPolygon()
drawPolyline()
drawOval()
drawArc()
```

Для них существуют соответствующие методы заполнения фигур, которые были предусмотрены в классе `Graphics` в версии JDK 1.0. В прикладном интерфейсе Java 2D API используется совершенно другая, объектно-ориентированная модель, где вместо методов применяются перечисленные ниже классы. Эти классы реализуют интерфейс `Shape` и рассматриваются в последующих подразделах.

```
Line2D
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
QuadCurve2D
CubicCurve2D
GeneralPath
```

11.3.2.1. Иерархия классов рисования фигур

Классы `Line2D`, `Rectangle2D`, `RoundRectangle2D`, `Ellipse2D` и `Arc2D` соответствуют методам `drawLine()`, `drawRectangle()`, `drawRoundRect()`, `drawOval()` и `drawArc()`. Для понятия "трехмерный прямоугольник" не предусмотрено соответствующего метода `draw3DRect()`. Но в прикладном интерфейсе Java 2D API поддерживаются два дополнительных класса для рисования кривых второго и третьего порядков, рассматриваемых далее. Для рисования многоугольников также не предусмотрено отдельного класса вроде `Polygon2D`, а предлагается класс `GeneralPath`, описывающий контуры, состоящие из линий и кривых второго и третьего порядков. Класс `GeneralPath` можно использовать для построения многоугольников, как поясняется далее.

Чтобы нарисовать фигуру, необходимо сначала создать объект класса, реализующего интерфейс `Shape`, а затем вызвать метод `draw()` из класса `Graphics2D`. Перечисленные ниже классы наследуют общий класс `RectangularShape`. Как известно, эллипсы и дуги не являются прямоугольниками, но их можно вписать в *ограничивающий* прямоугольник (рис. 11.32).

```
Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D
```

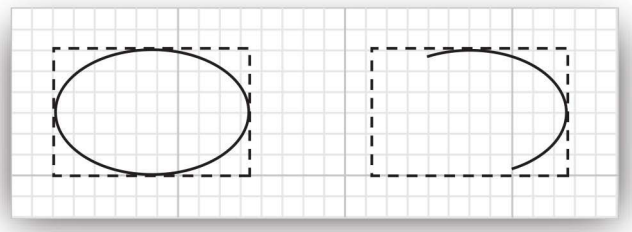


Рис. 11.32. Эллипс и дугу можно вписать в ограничивающий прямоугольник

У каждого из классов, название которого оканчивается на `2D`, имеются два подкласса для задания координат в виде числовых значений типа `float` и `double`. Так, в первом томе настоящего издания уже встречались обозначения `Rectangle2D.Float` и `Rectangle2D.Double`. Такая же схема используется для обозначения других классов, например `Arc2D.Float` и `Arc2D.Double`.

Для внутреннего представления координат во всех классах, предназначенных для построения графики, используются числовые данные типа `float`, поскольку они требуют меньше места для хранения, чем числовые данные типа `double`. К тому же они поддерживают достаточную точность для геометрических расчетов. Но в языке Java обработка числовых данных типа `float` выполняется очень громоздкими и неуклюжими средствами. Поэтому большинство методов из классов для построения графики принимают параметры типа `double` и возвращают значения типа `double`. Выбирать тип числовых данных (`float` или `double`) и соответствующий конструктор класса приходится только при создании двухмерного объекта, как показано в следующем примере кода:

```
var floatRect = new Rectangle2D.Float(5F, 10F, 7.5F, 15F);
var doubleRect = new Rectangle2D.Double(5, 10, 7.5, 15);
```

Классы *Xxx*`2D.Float` и *Xxx*`2D.Double` являются производными от классов *Xxx*`2D`. После создания объекта нет никакой нужды запоминать подклассы, а можно просто сохранить созданный объект в переменной суперкласса, как показано в приведенном выше примере кода.

Судя по названиям классов *Xxx*`2D.Float` и *Xxx*`2D.Double`, можно сделать вывод, что они являются внутренними для классов `Xxx2D`. Это простое синтаксическое правило позволяет избежать чрезмерного увеличения длины имен внешних классов. Наконец, предусмотрен класс `Point2D`, описывающий точку с координатами *x* и *y*. Точки полезны для определения фигур, но сами они не являются фигурами.

На рис. 11.33 схематически представлены отношения наследования между классами рисования фигур без указания подклассов `Double` и `Float`. Устаревшие классы, унаследованные из прежних версий JDK, выделены серым цветом.

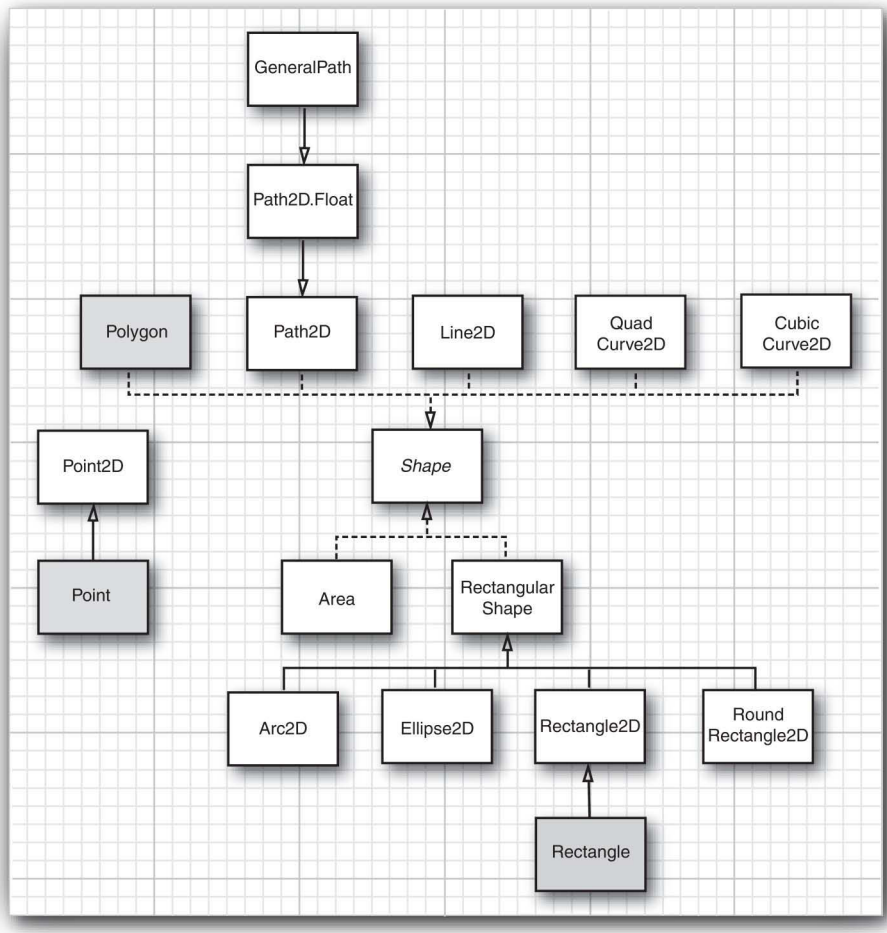


Рис. 11.33. Отношения наследования между классами рисования фигур

11.3.2.2. Применение классов рисования фигур

О классах `Line2D`, `Rectangle2D` и `Ellipse2D` уже упоминалось в главе 10 первого тома настоящего издания, а в этом разделе речь пойдет о том, как пользоваться классами для построения остальных двухмерных фигур. В частности, для построения прямоугольника со скругленными углами в конструкторе класса `RoundRectangle2D` следует указать координаты верхнего левого угла, ширину, высоту и размеры области скругления углов по осям *x* и *y* (рис. 11.34). Например, в результате следующего вызова конструктора данного класса создается прямоугольник с радиусом скругления углов, равным `20`:

```
var r = new RoundRectangle2D.Double(
        150, 200, 100, 50, 20, 20);
```

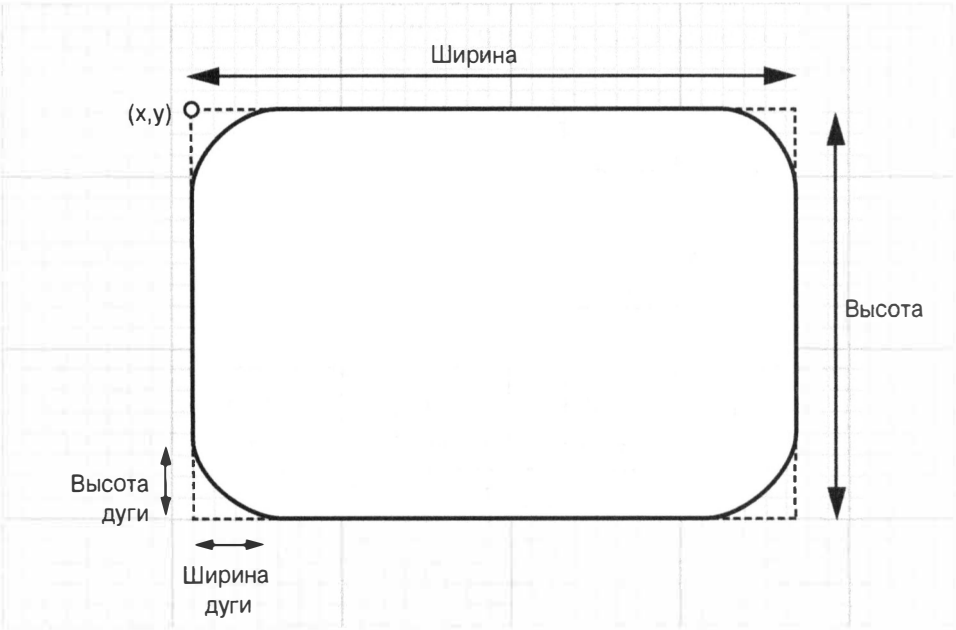


Рис. 11.34. Построение прямоугольника со скругленными углами средствами класса **`RoundRectangle2D`**

Для построения дуги следует указать ограничивающий прямоугольник, начальный угол, угол разворота дуги (рис. 11.35), а также один из видов замыкания дуги: `Arc2D.OPEN`, `Arc2D.PIE` или `Arc2D.CHORD`. Ниже приведен пример построения дуги, а на рис. 11.36 — виды замыкания дуги.

```
var a = new Arc2D(x, y, width, height, startAngle,
                  arcAngle, closureType);
```

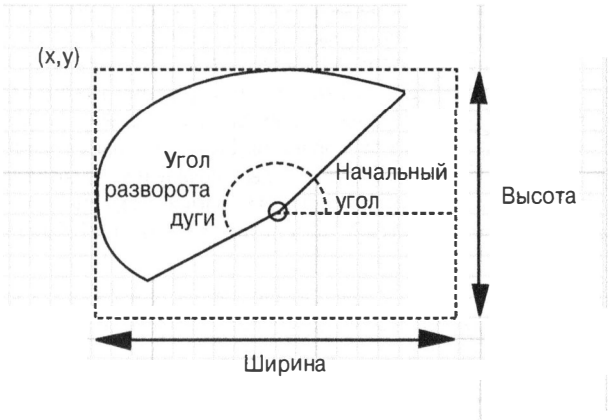


Рис. 11.35. Построение эллиптической дуги

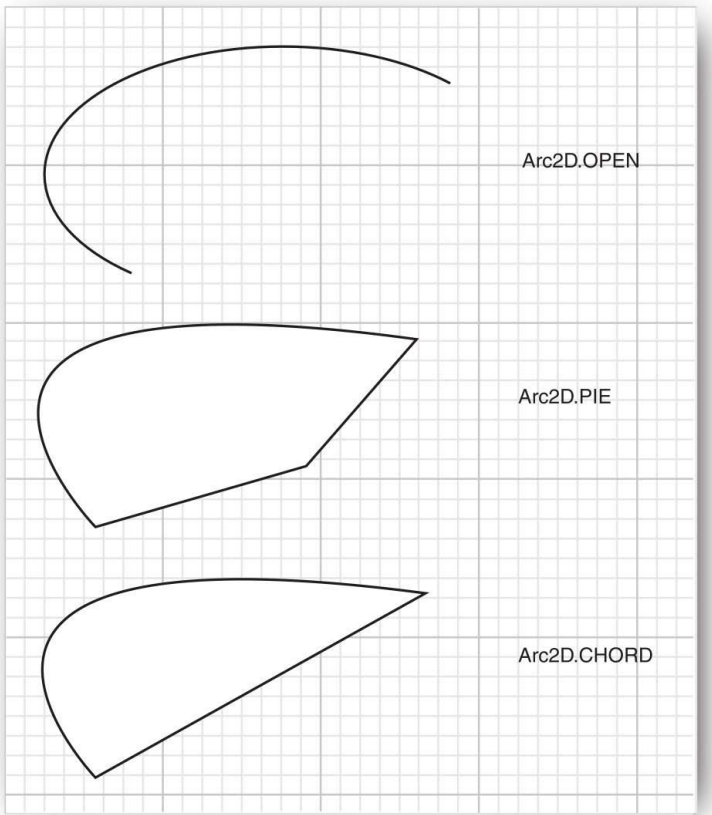


Рис. 11.36. Виды замыкания дуги

ВНИМАНИЕ! Если дуга имеет эллиптическую форму, рассчитать ее углы не так-то просто. По этому поводу в документации на прикладной интерфейс Java 2D API заявляется следующее: "Углы задаются относительно ограничивающего прямоугольника таким образом, чтобы линия, проведенная из центра эллипса к правому верхнему углу ограничивающего прямоугольника, образовывала угол 45° с обеими его сторонами. Если же обрамляющий прямоугольник заметно длиннее по одной оси, чем по другой, углы к началу и концу отрезка дуги будут скашиваться вдоль более длинной оси ограничивающей рамки". Но, к сожалению, в документации ничего не говорится о том, как рассчитывать такое "скашивание". Поэтому покажем, как это делается на практике.

Допустим, центр дуги находится в точке начала отсчета, а точка с координатами (*x*, *y*) — на самой дуге. Угол скашивания в этом случае рассчитывается следующим образом:

```
skewedAngle = Math.toDegrees(Math.atan2(-y * height, x * width));
```

В итоге получается значение в пределах от **–180** до **180**. Подобным образом рассчитываются сначала начальный и конечный углы скашивания, а затем их разность. Если получится отрицательное значение, то прибавляется значение **360**. Далее остается лишь предоставить конструктору дуги значения начального угла и разности двух углов скашивания в качестве параметров.

Запустив на выполнение пример программы, исходный код которой приведен в конце этого раздела, можете сами убедиться, что описанный выше порядок расчета углов скашивания действительно дает правильные значения параметров для конструктора дуги, как показано далее на рис. 11.39.

В прикладном интерфейсе Java 2D API предусмотрены средства для построения кривых *второго* порядка (квадратичных) и *третьего* порядка (кубических). Здесь не рассматриваются математические аспекты построения этих кривых, но для демонстрации их возможностей предлагается запустить на выполнение программу из листинга 11.15. Как показано на рис. 11.37 и 11.38, кривые второго и третьего порядка определяются двумя *конечными точками* и одной или двумя *управляющими точками* (для кривых второго и третьего порядка соответственно). При перемещении управляющих точек изменяется форма кривых. Для построения кривых второго и третьего порядка задаются координаты конечных и управляющих точек, как показано в приведенном ниже примере кода.

```
var q = new QuadCurve2D.Double(startX, startY, control1X,
                               control1Y, endX, endY);
var c = new CubicCurve2D.Double(startX, startY, control1X,
                                control1Y, control2X,
                                control2Y, endX, endY);
```

Кривые второго порядка не очень удобны, поэтому они редко применяются на практике, в отличие от кривых третьего порядка (например, кривых Безье, вычерчиваемых средствами класса `CubicCurve2D`). Сочетая несколько кривых третьего порядка таким образом, чтобы в точках соединения совпадали углы их наклона, можно строить сложные поверхности. Более подробно об этом можно узнать в книге *Computer Graphics: Principles and Practice, Third Edititon* Джеймса Фоли, Андриса ван Дама, Стивена Фэйнера и др. (James D. Foley, Andries van Dam, Steven K. Feiner et al.; издательство Addison Wesley, 2013 г.).

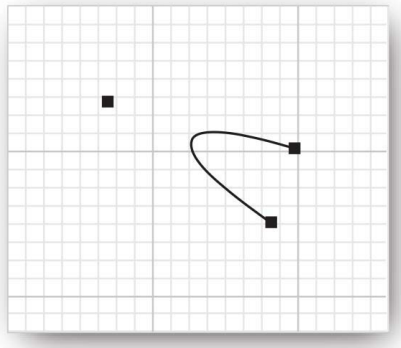


Рис. 11.37. Кривая второго порядка

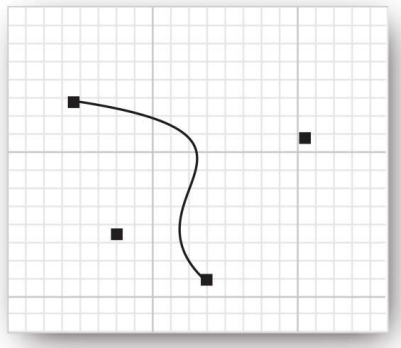


Рис. 11.38. Кривая третьего порядка

Для построения и сохранения произвольной кривой в виде последовательности линейных отрезков, кривых второго и третьего порядка предусмотрен класс `GeneralPath`. С этой целью методу `moveTo()` следует передать координаты первой точки контура строящейся кривой:

```
var path = new GeneralPath();
path.moveTo(10, 20);
```

Далее контур строящейся кривой расширяется с помощью метода `lineTo()`, `quadTo()` или `curveTo()`. Эти методы продолжают контур строящейся кривой линией, кривой второго порядка или кривой третьего порядка соответственно. При вызове метода `lineTo()` следует указать конечную точку, а при вызове двух других методов построения кривых — сначала управляющие точки и затем конечную точку, как показано ниже. Наконец, контур строящейся кривой замыкается с помощью метода `closePath()`, где проводится линия в обратном направлении к начальной точке данного контура.

```
path.lineTo(20, 30);
path.curveTo(control1X, control1Y, control2X, control2Y,
             endX, endY);
```

Чтобы построить многоугольник, следует вызвать метод `moveTo()` для перехода к первой его вершине, а затем несколько раз вызвать метод `lineTo()` для соединения отрезками линии остальных вершин многоугольника и вызвать метод `closePath()`, чтобы замкнуть многоугольник. Более подробно этот процесс демонстрируется далее в примере программы из листинга 11.15. Общий контур вычерчиваемой фигуры совсем не обязательно должен быть соединенным. Это означает, что метод `moveTo()` можно вызвать в любой удобный момент, чтобы начать построение нового отрезка контура.

Наконец, к общему контуру можно добавить произвольные объекты типа `Shape`, используя для этой цели метод `append()`. При этом контур присоединяемой фигуры добавляется в конце общего контура вычерчиваемой фигуры. Второй параметр этого метода принимает логическое значение `true`, если новую фигуру необходимо соединить с последней точкой общего контура, а иначе — логическое значение `false`. Например, в приведенном ниже фрагменте кода контур прямоугольника добавляется к общему контуру вычерчиваемой фигуры, не соединяясь с этим контуром в его конце.

```
Rectangle2D r = . . .;
path.append(r, false);
```

При следующем вызове метода `append()` сначала проводится прямая линия, соединяющая конечную точку общего контура с начальной точкой прямоугольника **`r`**, а затем контур прямоугольника добавляется к общему контуру вычерчиваемой фигуры:

```
path.append(r, true);
```

На рис. 11.37 и 11.38 показаны результаты выполнения примера программы из листинга 11.15. В рабочем окне этой программы находится раскрывающийся список для выбора следующих видов вычерчиваемых фигур.

- Прямые линии.
- Прямоугольники, в том числе скругленные, а также эллипсы.
- Дуги (помимо самой дуги, отображаются контуры ограничивающего прямоугольника, маркер начального угла и угол разворота дуги).
- Многоугольники (вычерчиваемые средствами класса `GeneralPath`).
- Кривые второго и третьего порядка.

Изменить расположение управляющих точек можно с помощью мыши. Обратите внимание на то, что при перемещении этих точек фигура автоматически перерисовывается. Рассматриваемая здесь программа имеет довольно сложную структуру, поскольку она оперирует многими фигурами и поддерживает их автоматическую перерисовку при перетаскивании управляющих точек.

Абстрактный суперкласс `ShapeMaker` инкапсулирует общие функции всех классов для построения фигур. У каждой фигуры имеются управляющие точки, которые пользователь может перемещать, а метод `getPointCount()` возвращает их количество. Для расчета конкретной формы фигуры, исходя из текущего положения управляющих точек, служит следующий абстрактный метод:

```
Shape makeShape(Point2D[] points)
```

Метод `toString()` возвращает имя класса, и благодаря этому объекты типа `ShapeMaker` могут быть просто внесены в комбинированный список типа `JComboBox`. Для активизации режима перетаскивания управляющих точек в классе `ShapePanel` следует предусмотреть обработку событий от мыши. Так, если кнопка мыши нажата над прямоугольным маркером управляющей точки, этот маркер перемещается при последующем движении мыши.

Большинство классов для построения фигур имеют очень простую структуру. Их методы `makeShape()` строят и возвращают требующуюся фигуру. Но для применения класса `ArcMaker` приходится дополнительно рассчитывать начальный и конечный углы скашивания, как пояснялось выше. Поэтому для демонстрации правильности выполненных расчетов возвращаемая фигура вычерчивается средствами класса `GeneralPath`, включая саму дугу, ограничивающий ее прямоугольник и линии, проведенные из центра дуги к управляющим точкам (рис. 11.39).

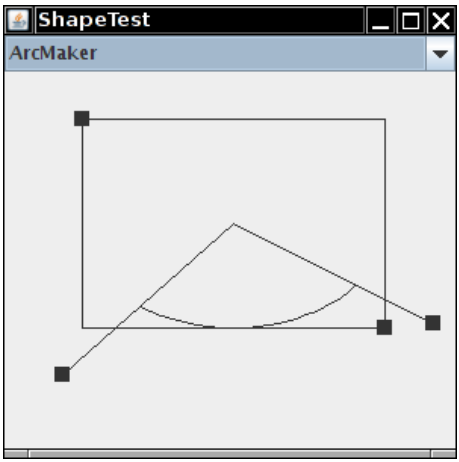


Рис. 11.39. Рабочее окно программы **ShapeTest** с построенной дугой и ее вспомогательными элементами

Листинг 11.15. Исходный код из файла **shape/ShapeTest.java**

```
1   package shape;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.awt.geom.*;
6   import java.util.*;
7   import javax.swing.*;
8
9   /**
10   * В этой программе демонстрируется построение
11   * различных двухмерных фигур
12   * @version 1.04 2018-05-01
13   * @author Cay Horstmann
14   */
```

```
15  public class ShapeTest
16  {
17    public static void main(String[] args)
18    {
19      EventQueue.invokeLater(() ->
20        {
21          var frame = new ShapeTestFrame();
22          frame.setTitle("ShapeTest");
23          frame.setDefaultCloseOperation(
24                JFrame.EXIT_ON_CLOSE);
25          frame.setVisible(true);
26        });
27    }
28  }
29
30  /**
31   * Этот фрейм содержит комбинированный список для
32   * выбора фигур, а также компонент для их рисования
33   */
34  class ShapeTestFrame extends JFrame
35  {
36    public ShapeTestFrame()
37    {
38      var comp = new ShapeComponent();
39      add(comp, BorderLayout.CENTER);
40      var comboBox = new JComboBox<ShapeMaker>();
41      comboBox.addItem(new LineMaker());
42      comboBox.addItem(new RectangleMaker());
43      comboBox.addItem(new RoundRectangleMaker());
44      comboBox.addItem(new EllipseMaker());
45      comboBox.addItem(new ArcMaker());
46      comboBox.addItem(new PolygonMaker());
47      comboBox.addItem(new QuadCurveMaker());
48      comboBox.addItem(new CubicCurveMaker());
49      comboBox.addActionListener(event ->
50        {
51          ShapeMaker shapeMaker = comboBox.getItemAt(
52                          comboBox.getSelectedIndex());
53          comp.setShapeMaker(shapeMaker);
54        });
55      add(comboBox, BorderLayout.NORTH);
56      comp.setShapeMaker((ShapeMaker)
57           comboBox.getItemAt(0));
58      pack();
59    }
60  }
61
62  /**
63   * Этот компонент рисует фигуру и дает пользователю
64   * возможность перемещать определяющие ее точки
65   */
66  class ShapeComponent extends JComponent
67  {
68    private static final Dimension PREFERRED_SIZE =
69                          new Dimension(300, 200);
70    private Point2D[] points;
```

```
71    private static Random generator = new Random();
72    private static int SIZE = 10;
73    private int current;
74    private ShapeMaker shapeMaker;
75
76    public ShapeComponent()
77    {
78    · addMouseListener(new MouseAdapter()
79       {
80         public void mousePressed(MouseEvent event)
81         {
82           Point p = event.getPoint();
83           for (int i = 0; i < points.length; i++)
84           {
85             double x = points[i].getX() - SIZE / 2;
86             double y = points[i].getY() - SIZE / 2;
87             var r = new Rectangle2D.Double(
88                         x, y, SIZE, SIZE);
89             if (r.contains(p))
90             {
91               current = i;
92               return;
93             }
94           }
95         }
96
97         public void mouseReleased(MouseEvent event)
98         {
99           current = -1;
100        }
101      });
102   addMouseMotionListener(new MouseMotionAdapter()
103     {
104       public void mouseDragged(MouseEvent event)
105       {
106         if (current == -1) return;
107         points[current] = event.getPoint();
108         repaint();
109       }
110     });
111   current = -1;
112   }
113
114   /**
115    * Устанавливает построитель фигур и инициализирует
116    * его произвольным рядом точек
117    * @param aShapeMaker Построитель фигур, определяющий
118    *                    фигуру по ряду исходных точек
119    */
120   public void setShapeMaker(ShapeMaker aShapeMaker)
121   {
122     shapeMaker = aShapeMaker;
123     int n = shapeMaker.getPointCount();
124     points = new Point2D[n];
125     for (int i = 0; i < n; i++)
126     {
```

```
127       double x = generator.nextDouble() * getWidth();
128       double y = generator.nextDouble() * getHeight();
129       points[i] = new Point2D.Double(x, y);
130     }
131     repaint();
132   }
133
134   public void paintComponent(Graphics g)
135   {
136     if (points == null) return;
137     var g2 = (Graphics2D) g;
138     for (int i = 0; i < points.length; i++)
139     {
140       double x = points[i].getX() - SIZE / 2;
141       double y = points[i].getY() - SIZE / 2;
142       g2.fill(new Rectangle2D.Double(x, y, SIZE, SIZE));
143     }
144
145     g2.draw(shapeMaker.makeShape(points));
146   }
147
148   public Dimension getPreferredSize()
149   { return PREFERRED_SIZE; }
150 }
151
152 /**
153  * Построитель фигур по ряду исходных точек.
154  * Конкретные подклассы должны возвращать
155  * форму из метода makeShape()
156  */
157 abstract class ShapeMaker
158 {
159   private int pointCount;
160
161   /**
162    * Конструирует построитель фигур
163    * @param ointCount Количество исходных точек,
164    *                  требующихся для определения
165    *                  данной фигуры
166    */
167   public ShapeMaker(int pointCount)
168   {
169     this.pointCount = pointCount;
170   }
171
172   /**
173    * Получает количество исходных точек, требующихся
174    * для определения данной фигуры
175    * @return Количество исходных точек
176    */
177   public int getPointCount()
178   {
179     return pointCount;
180   }
181
182   /**
```

```
183    * Строит фигуру по заданному ряду исходных точек
184    * @param p Исходные точки, определяющие фигуру
185    * @return Фигура, определяемая исходными точками
186    */
187   public abstract Shape makeShape(Point2D[] p);
188
189   public String toString()
190   {
191     return getClass().getName();
192   }
193 }
194
195 /**
196  * Проводит линию, соединяющую две заданные точки
197  */
198 class LineMaker extends ShapeMaker
199 {
200   public LineMaker()
201   {
202     super(2);
203   }
204
205   public Shape makeShape(Point2D[] p)
206   {
207     return new Line2D.Double(p[0], p[1]);
208   }
209 }
210
211 /**
212  * Строит прямоугольник, соединяющий две
213  * заданные угловые точки
214  */
215 class RectangleMaker extends ShapeMaker
216 {
217   public RectangleMaker()
218   {
219     super(2);
220   }
221
222   public Shape makeShape(Point2D[] p)
223   {
224     var s = new Rectangle2D.Double();
225     s.setFrameFromDiagonal(p[0], p[1]);
226     return s;
227   }
228 }
229
230 /**
231  * Строит прямоугольник со скругленными углами,
232  * соединяющий две заданные угловые точки
233  */
234 class RoundRectangleMaker extends ShapeMaker
235 {
236   public RoundRectangleMaker()
237   {
238     super(2);
```

```
239    }
240
241    public Shape makeShape(Point2D[] p)
242    {
243      var s = new RoundRectangle2D.Double(
244                      0, 0, 0, 0, 20, 20);
245      s.setFrameFromDiagonal(p[0], p[1]);
246      return s;
247    }
248 }
249
250 /**
251  * Строит эллипс, вписанный в ограничивающий
252  * прямоугольник с двумя заданными угловыми точками
253  */
254 class EllipseMaker extends ShapeMaker
255 {
256    public EllipseMaker()
257    {
258      super(2);
259    }
260
261    public Shape makeShape(Point2D[] p)
262    {
263      var s = new Ellipse2D.Double();
264      s.setFrameFromDiagonal(p[0], p[1]);
265      return s;
266    }
267 }
268
269 /**
270  * Строит дугу, вписанную в ограничивающий прямоугольник
271  * с двумя заданными угловыми точками. Начальный и
272  * конечный углы дуги задают линии, проведенные из
273  * центра ограничивающего прямоугольника в две заданные
274  * точки. Для демонстрации правильности расчета углов
275  * возвращаемая фигура состоит из дуги, ограничивающего
276  * прямоугольника и линий, образующих эти углы
277  */
278 class ArcMaker extends ShapeMaker
279 {
280    public ArcMaker()
281    {
282      super(4);
283    }
284
285    public Shape makeShape(Point2D[] p)
286    {
287      double centerX = (p[0].getX() + p[1].getX()) / 2;
288      double centerY = (p[0].getY() + p[1].getY()) / 2;
289      double width = Math.abs(p[1].getX() - p[0].getX());
290      double height = Math.abs(p[1].getY() - p[0].getY());
291
292      double skewedStartAngle = Math.toDegrees(
293            Math.atan2(-(p[2].getY() - centerY) * width,
294            (p[2].getX() - centerX) * height));
```

```
295     double skewedEndAngle = Math.toDegrees(
296            Math.atan2(-(p[3].getY() - centerY) * width,
297            (p[3].getX() - centerX) * height));
298     double skewedAngleDifference =
299            skewedEndAngle - skewedStartAngle;
300     if (skewedStartAngle < 0)
301       skewedStartAngle += 360;
302     if (skewedAngleDifference < 0)
303       skewedAngleDifference += 360;
304
305     var s = new Arc2D.Double(
306                 0, 0, 0, 0, skewedStartAngle,
307                 skewedAngleDifference, Arc2D.OPEN);
308     s.setFrameFromDiagonal(p[0], p[1]);
309
310     var g = new GeneralPath();
311     g.append(s, false);
312     var r = new Rectangle2D.Double();
313     r.setFrameFromDiagonal(p[0], p[1]);
314     g.append(r, false);
315     var center = new Point2D.Double(centerX, centerY);
316     g.append(new Line2D.Double(center, p[2]), false);
317     g.append(new Line2D.Double(center, p[3]), false);
318     return g;
319   }
320 }
321
322 /**
323  * Строит многоугольник, определяемый шестью
324  * угловыми точками
325  */
326 class PolygonMaker extends ShapeMaker
327 {
328   public PolygonMaker()
329   {
330     super(6);
331   }
332
333   public Shape makeShape(Point2D[] p)
334   {
335     var s = new GeneralPath();
336     s.moveTo((float) p[0].getX(), (float) p[0].getY());
337     for (int i = 1; i < p.length; i++)
338       s.lineTo((float) p[i].getX(),
339                (float) p[i].getY());
340     s.closePath();
341     return s;
342   }
343 }
344
345 /**
346  * Строит кривую второго порядка, определяемую двумя
347  * конечными точками и одной управляющей точкой
348  */
349 class QuadCurveMaker extends ShapeMaker
350 {
```

```
351   public QuadCurveMaker()
352   {
353     super(3);
354   }
355
356   public Shape makeShape(Point2D[] p)
357   {
358     return new QuadCurve2D.Double(
359                p[0].getX(), p[0].getY(), p[1].getX(),
360                p[1].getY(), p[2].getX(), p[2].getY());
361   }
362 }
363
364 /**
365  * Строит кривую третьего порядка, определяемую двумя
366  * конечными точками и двумя управляющими точками
367  */
368 class CubicCurveMaker extends ShapeMaker
369 {
370   public CubicCurveMaker()
371   {
372     super(4);
373   }
374
375   public Shape makeShape(Point2D[] p)
376   {
377     return new CubicCurve2D.Double(
378                p[0].getX(), p[0].getY(), p[1].getX(),
379                p[1].getY(), p[2].getX(), p[2].getY(),
380                p[3].getX(), p[3].getY());
381   }
382 }
```

java.awt.geom.RoundRectangle2D.Double 1.2

- **RoundRectangle2D.Double(double *x*, double *y*, double *width*, double *height*, double *arcWidth*, double *arcHeight*)**

  Строит прямоугольник со скругленными углами, исходя из заданных размеров ограничивающего прямоугольника и дуги скругления. Наглядное представление о параметрах ***arcWidth*** и ***arcHeight*** дает рис. 11.34.

java.awt.geom.Arc2D.Double 1.2

- **Arc2D.Double(double *x*, double *y*, double *w*, double *h*, double *startAngle*, double *arcAngle*, int *type*)**

  Строит дугу, исходя из заданного ограничивающего прямоугольника, начального и конечного угла, также вида замыкания дуги. Наглядное представление о параметрах ***startAngle*** и ***arcAngle*** дает рис. 11.35. Что касается параметра ***type***, то он может принимать значение одной из следующих констант: **Arc2D.OPEN**, **Arc2D.PIE** или **Arc2D.CHORD**.

java.awt.geom.QuadCurve2D.Double 1.2

- **QuadCurve2D.Double(double *x1*, double *y1*, double *ctrlx*, double *ctrly*, double *x2*, double *y2*)**
  Строит кривую второго порядка по начальной, управляющей и конечной точкам.

java.awt.geom.CubicCurve2D.Double 1.2

- **CubicCurve2D.Double(double *x1*, double *y1*, double *ctrlx1*, double *ctrly1*, double *ctrlx2*, double *ctrly2*, double *x2*, double *y2*)**
  Создает кривую третьего порядка по начальной, двум контрольным и конечной точкам.

java.awt.geom.GeneralPath 1.2

- **GeneralPath()**
  Строит пустой общий контур.

java.awt.geom.Path2D.Float 6

- **void moveTo(float *x*, float *y*)**
  Строит *текущую точку* с координатами ***x*** и ***y***, т.е. начальную точку следующего отрезка линии.
- **void lineTo(float *x*, float *y*)**
- **void quadTo(float *ctrlx*, float *ctrly*, float *x*, float *y*)**
- **void curveTo(float *ctrlx1*, float *ctrly1*, float *ctrlx2*, float *ctrly2*, float *x*, float *y*)**
  Рисуют прямую линию, кривую второго или третьего порядка от текущей до конечной точки с координатами ***x*** и ***y***, после чего конечная точка становится текущей.

java.awt.geom.Path2D 6

- **void append(Shape *s*, boolean *connect*)**
  Присоединяет контур заданной фигуры к общему контуру. Если параметр ***connect*** принимает логическое значение **true**, текущая точка общего контура соединяется прямой линией с начальной точкой присоединяемой фигуры.
- **void closePath()**
  Замыкает контур, рисуя прямую линию от текущей точки к первой точке контура.

11.3.3. Участки

В предыдущем разделе обсуждались средства построения сложных фигур с помощью прямых линий и кривых второго и третьего порядка. Используя

достаточное количество линий и кривых, можно нарисовать практически любую фигуру. Например, контуры символов в шрифтах, которые отображаются на экране и на отпечатке, состоят из прямых линий и кривых третьего порядка.

Но иногда описать фигуру легче, составляя ее из *участков*, например, прямоугольных, многоугольных или овальных. В прикладном интерфейсе Java 2D API поддерживаются четыре метода, выполняющие геометрические операции построения нового участка из двух исходных участков. Результаты выполнения этих операций приведены на рис. 11.40.

- Метод `add()` составляет участок, который содержит все точки первого или второго исходного участка. Это операция геометрического сложения участков.
- Метод `subtract()` составляет участок, который содержит все точки первого исходного участка, не входящие во второй исходный участок. Это операция геометрического вычитания участков.
- Метод `intersect()` составляет участок, который содержит только точки, принадлежащие одновременно первому и второму исходному участкам. Это операция геометрического пересечения участков.
- Метод `exclusiveOr()` составляет участок, который содержит все точки, принадлежащие первому или второму исходному участку, но не обоим вместе. Это операция геометрического исключения участков.

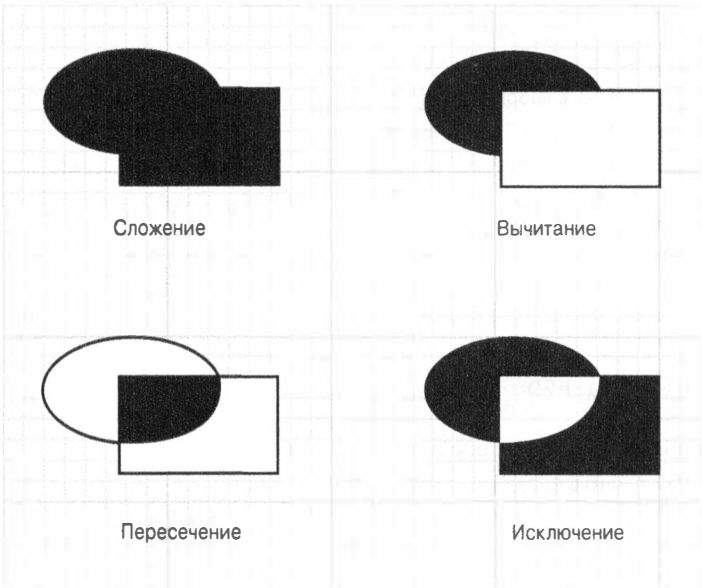


Рис. 11.40. Геометрические операции построения нового участка из двух исходных участков

Чтобы построить участок сложной геометрической формы, необходимо создать сначала исходный участок:

```
var a = new Area();
```

Затем этот участок следует объединить с фигурой требующейся формы, выполнив одну из упомянутых выше геометрических операций:

```
a.add(new Rectangle2D.Double(. . .));
a.subtract(path);
. . .
```

Класс `Area` для построения участков реализует интерфейс `Shape`. В процессе построения участка можно очертить его границы с помощью метода `draw()` или заполнить внутреннюю его часть с помощью метода `fill()` из класса `Graphics2D`.

java.awt.geom.Area

- **void add(Area *other*)**
- **void subtract(Area *other*)**
- **void intersect(Area *other*)**
- **void exclusiveOr(Area *other*)**

  Выполняют геометрические операции построения нового участка, объединяя текущий участок с другим участком, определяемым параметром ***other***. Получающийся в итоге участок становится текущим.

11.3.4. Обводка

Метод `draw()` из класса `Graphics2D` рисует границу фигуры, используя выбранный в настоящий момент вид *обводки*. По умолчанию для обводки выбирается сплошная линия толщиной в один пиксель. Для изменения вида обводки предусмотрен метод `setStroke()`, которому в качестве параметра передается экземпляр класса, реализующего интерфейс `Stroke`. В прикладном интерфейсе Java 2D API определен только один такой класс — `BasicStroke`. В этом разделе рассматриваются функциональные возможности данного класса.

Для обводки можно задать линию произвольной толщины. Например, для выбора линии толщиной 10 пикселей и последующей обводки служит следующий код:

```
g2.setStroke(new BasicStroke(10.0F));
g2.draw(new Line2D.Double(. . .));
```

Если толщина линии обводки превышает один пиксель, то для формирования концов линии можно воспользоваться одним из перечисленных ниже стилей (рис. 11.41).

- **Торцевой конец линии.** Линия обводки обрывается в конечной точке.
- **Скругленный конец линии.** В конце линии обводки добавляется полукруг.
- **Квадратный конец линии.** В конце линии обводки добавляется полуквадрат.

Для соединения двух толстых линий обводки можно также указать один из следующих стилей (рис. 11.42).

- **Косой стык.** Соединяет линии обводки по прямой линии, перпендикулярной биссектрисе угла между ними.
- **Скругленный стык.** Соединяет линии обводки, образуя скругленный конец линии.
- **Угловой стык.** Соединяет линии обводки, образуя клинообразный конец линии.

Рис. 11.41. Стили окончания линии обводки

Рис. 11.42. Стили соединения линий обводки

Если две линии сходятся в угловом стыке под очень малым углом, тогда вместо него используется косой стык, предотвращающий образование чрезмерно длинных клиньев. А определяет такой переход между линиями *предел среза*, который формально обозначает отношение расстояния между внутренним и внешним углами клина к его ширине. По умолчанию предел среза равен 10, что соответствует углу около 11°.

```
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_ROUND,
             BasicStroke.JOIN_ROUND));
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT,
      BasicStroke.JOIN_MITER, 15.0F /* предел среза */));
```

Наконец, для пунктирных линий обводки можно выбрать конкретный *пунктир*. В примере программы из листинга 11.16 используется пунктир, соответствующий сигналу SOS в азбуке Морзе. Пунктир задается в виде массива типа `float[]`, который содержит длины рисуемых и пробельных отрезков обводки (рис. 11.43).

При создании объекта типа `BasicStroke`, помимо самого пунктира, указывается также *фаза пунктира*, т.е. место начала пунктира на линии обводки. Обычно

фаза пунктира устанавливается равной нулю. В приведенном ниже примере кода показано, каким образом задается пунктирный стиль линий обводки.

```
float[] dashPattern = { 10, 10, 10, 10, 10, 10, 30, 10, 30, ... };
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT,
      BasicStroke.JOIN_MITER, 10.0F /* предел среза */,
      dashPattern, 0 /* фаза пунктира */));
```

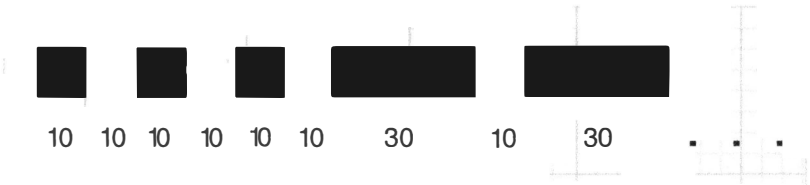


Рис. 11.43. Пунктир для обводки

НА ЗАМЕТКУ! Стили окончания линий распространяются и на все штрихи пунктира.

В примере программы из листинга 11.16 предоставляется возможность указать стиль окончания линий обводки, стиль их соединения и пунктир (рис. 11.44). Кроме того, концы линий обводки можно перемещать, чтобы проверить действенность предела среза. Для этого следует выбрать угловой стык, а затем перетащить мышью отрезок линии обводки таким образом, чтобы две линии образовали очень острый угол. По достижения предела среза угловой стык автоматически превращается в косой.

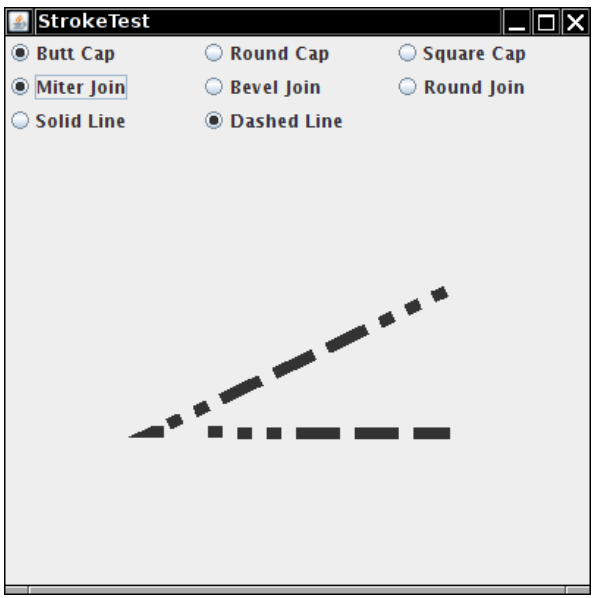


Рис. 11.44. Рабочее окно программы `StrokeTest`

Рассматриваемая здесь программа похожа на программу из листинга 11.15. Обработчик событий от мыши реагирует на щелчок кнопкой мыши в конечной точке линии обводки и при перемещении курсора будет передвигать вслед за ним конец линии обводки. Группа кнопок-переключателей уведомляет о выборе пользователем стиля окончания и соединения линии обводки, а также сплошной или пунктирной линии. Метод `paintComponent()` из класса `StrokePanel` используется для построения общего контура типа `GeneralPath` из двух отрезков, соединяющих три точки, которые пользователь может перемещать с помощью мыши. Затем создается объект типа `BasicStroke`, исходя из выбора, сделанного пользователем. Наконец, линия обводки рисуется по сформированному контуру.

Листинг 11.16. Исходный код из файла **`stroke/StrokeTest.java`**

```
1  package stroke;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.awt.geom.*;
6  import javax.swing.*;
7
8  /**
9   * В этой программе демонстрируются
10  * различные виды обводки
11  * @version 1.05 2018-05-01
12  * @author Cay Horstmann
13  */
14 public class StrokeTest
15 {
16   public static void main(String[] args)
17   {
18     EventQueue.invokeLater(() ->
19       {
20         var frame = new StrokeTestFrame();
21         frame.setTitle("StrokeTest");
22         frame.setDefaultCloseOperation(
23               JFrame.EXIT_ON_CLOSE);
24         frame.setVisible(true);
25       });
26   }
27 }
28
29 /**
30  * В этом фрейме пользователь может выбрать стиль
31  * окончания и соединения линий обводки, а также
32  * увидеть получающуюся в итоге линию обводки
33  */
34 class StrokeTestFrame extends JFrame
35 {
36   private StrokeComponent canvas;
37   private JPanel buttonPanel;
38
39   public StrokeTestFrame()
40   {
```

```
41      canvas = new StrokeComponent();
42      add(canvas, BorderLayout.CENTER);
43
44      buttonPanel = new JPanel();
45      buttonPanel.setLayout(new GridLayout(3, 3));
46      add(buttonPanel, BorderLayout.NORTH);
47
48      var group1 = new ButtonGroup();
49      makeCapButton("Butt Cap", BasicStroke.CAP_BUTT,
50                    group1);
51      makeCapButton("Round Cap", BasicStroke.CAP_ROUND,
52                    group1);
53      makeCapButton("Square Cap", BasicStroke.CAP_SQUARE,
54                    group1);
55
56      var group2 = new ButtonGroup();
57      makeJoinButton("Miter Join", BasicStroke.JOIN_MITER,
58                     group2);
59      makeJoinButton("Bevel Join", BasicStroke.JOIN_BEVEL,
60                    group2);
61      makeJoinButton("Round Join", BasicStroke.JOIN_ROUND,
62                     group2);
63
64      var group3 = new ButtonGroup();
65      makeDashButton("Solid Line", false, group3);
66      makeDashButton("Dashed Line", true, group3);
67    }
68
69    /**
70     * Создает кнопку-переключатель для выбора
71     * стиля окончания линии
72     * @param label Метка кнопки-переключателя
73     * @param style Стиль окончания линии
74     * @param group Группа кнопок-переключателей
75     */
76    private void makeCapButton(String label,
77                  final int style, ButtonGroup group)
78    {
79      // выбрать первую кнопку-переключатель в группе
80      boolean selected = group.getButtonCount() == 0;
81      var button = new JRadioButton(label, selected);
82      buttonPanel.add(button);
83      group.add(button);
84      button.addActionListener(event ->
85            canvas.setCap(style));
86      pack();
87    }
88
89    /**
90     * Создает кнопку-переключатель для выбора
91     * стиля соединения линий
92     * @param label Метка кнопки-переключателя
93     * @param style Стиль соединения линий
94     * @param group Группа кнопок-переключателей
95     */
96    private void makeJoinButton(String label,
```

```
97                final int style, ButtonGroup group)
98    {
99      // выбрать первую кнопку-переключатель в группе
100     boolean selected = group.getButtonCount() == 0;
101     var button = new JRadioButton(label, selected);
102     buttonPanel.add(button);
103     group.add(button);
104     button.addActionListener(event ->
105           canvas.setJoin(style));
106   }
107
108   /**
109    * Создает кнопку-переключатель для выбора
110    * сплошных или пунктирных линий обводки
111    * @param label Метка кнопки-переключателя
112    * @param style Принимает логическое значение false,
113    *              если линия сплошная, или логическое
114    *              значение true, если линия пунктирная
115    */
116   private void makeDashButton(String label,
117                final boolean style, ButtonGroup group)
118   {
119     // выбрать первую кнопку-переключатель в группе
120     boolean selected = group.getButtonCount() == 0;
121     var button = new JRadioButton(label, selected);
122     buttonPanel.add(button);
123     group.add(button);
124     button.addActionListener(event ->
125           canvas.setDash(style));
126   }
127 }
128
129 /**
130  * Этот компонент рисует две соединенные линии,
131  * используя разные объекты обводки и давая
132  * пользователю возможность перетаскивать три точки,
133  * определяющие линии обводки
134  */
135 class StrokeComponent extends JComponent
136 {
137   private static final Dimension PREFERRED_SIZE =
138                        new Dimension(400, 400);
139   private static int SIZE = 10;
140
141   private Point2D[] points;
142   private int current;
143   private float width;
144   private int cap;
145   private int join;
146   private boolean dash;
147
148   public StrokeComponent()
149   {
150     addMouseListener(new MouseAdapter()
151       {
152         public void mousePressed(MouseEvent event)
```

```
153         {
154           Point p = event.getPoint();
155           for (int i = 0; i < points.length; i++)
156           {
157             double x = points[i].getX() - SIZE / 2;
158             double y = points[i].getY() - SIZE / 2;
159             var r = new Rectangle2D.Double(
160                         x, y, SIZE, SIZE);
161             if (r.contains(p))
162             {
163               current = i;
164               return;
165             }
166           }
167         }
168
169         public void mouseReleased(MouseEvent event)
170         {
171           current = -1;
172         }
173       });
174
175     addMouseMotionListener(new MouseMotionAdapter()
176       {
177         public void mouseDragged(MouseEvent event)
178         {
179           if (current == -1) return;
180           points[current] = event.getPoint();
181           repaint();
182         }
183       });
184
185     points = new Point2D[3];
186     points[0] = new Point2D.Double(200, 100);
187     points[1] = new Point2D.Double(100, 200);
188     points[2] = new Point2D.Double(200, 200);
189     current = -1;
190     width = 8.0F;
191   }
192
193   public void paintComponent(Graphics g)
194   {
195     var g2 = (Graphics2D) g;
196     var path = new GeneralPath();
197     path.moveTo((float) points[0].getX(),
198                 (float) points[0].getY());
199     for (int i = 1; i < points.length; i++)
200       path.lineTo((float) points[i].getX(),
201                   (float) points[i].getY());
202     BasicStroke stroke;
203     if (dash)
204     {
205       float miterLimit = 10.0F;
206       float[] dashPattern = { 10F, 10F, 10F, 10F, 10F,
207                     10F, 30F, 10F, 30F, 10F, 30F, 10F,
208                     10F, 10F, 10F, 10F, 10F, 30F };
```

```
209       float dashPhase = 0;
210       stroke = new BasicStroke(width, cap, join,
211                   miterLimit, dashPattern, dashPhase);
212     }
213     else stroke = new BasicStroke(width, cap, join);
214     g2.setStroke(stroke);
215     g2.draw(path);
216   }
217
218   /**
219    * Устанавливает стиль соединения линий
220    * @param j Стиль соединения линий
221    */
222   public void setJoin(int j)
223   {
224     join = j;
225     repaint();
226   }
227
228   /**
229    * Устанавливает стиль окончания линий
230    * @param c Стиль окончания линий
231    */
232   public void setCap(int c)
233   {
234     cap = c;
235     repaint();
236   }
237
238   /**
239    * Устанавливает сплошные или пунктирные линии
240    * @param d Принимает логическое значение false,
241    *          если линии сплошные, или логическое
242    *          значение true, если линии пунктирные
243    */
244   public void setDash(boolean d)
245   {
246     dash = d;
247     repaint();
248   }
249
250   public Dimension getPreferredSize()
251   { return PREFERRED_SIZE; }
252 }
```

java.awt.Graphics2D 1.2

- **void setStroke(Stroke *s*)**

  Устанавливает в качестве обводки для данного графического контекста указанный объект, класс которого реализует интерфейс **Stroke**.

java.awt.BasicStroke 1.2

- **BasicStroke(float *width*)**
- **BasicStroke(float *width*, int *cap*, int *join*)**
- **BasicStroke(float *width*, int *cap*, int *join*, float *miterlimit*)**
- **BasicStroke(float *width*, int *cap*, int *join*, float *miterlimit*, float[] *dash*, float *dashPhase*)**

Конструируют объект обводки с заданными свойствами.

Параметры:		
	width	Толщина обводки
	cap	Стиль окончания линии: **CAP_BUTT**, **CAP_ROUND** или **CAP_SQUARE**
	join	Стиль соединения линий: **JOIN_BEVEL**, **JOIN_MITER** или **JOIN_ROUND**
	miterlimit	Предел среза (в градусах), ниже которого угловой стык превращается в косой
	dash	Пунктир в виде массива длин отображаемых и пробельных отрезков обводки
	dashPhase	Фаза пунктира — длина отрезка, предшествующего первой точке линии обводки, при условии, что пунктир уже применяется до этой точки

11.3.5. Раскраска

Внутренняя часть фигуры заполняется путем *раскраски*. Для установки стиля раскраски служит объект, класс которого реализует интерфейс `Paint`. Этот объект передается методу `setPaint()`. В прикладном интерфейсе Java 2D API для этой цели предусмотрены следующие классы.

- Класс `Color` служит для заполнения фигуры сплошным цветом. Для этого достаточно вызвать метод `setPaint()`, указав объект типа `Color`:

```
g2.setPaint(Color.red);
```

- Класс `GradientPaint` служит для заполнения фигуры градиентом, т.е. цветом, постепенно изменяющим свой оттенок (рис. 11.45).
- Класс `TexturePaint` служит для заполнения фигуры текстурой, т.е. повторяющимся рисунком (рис. 11.46).

Рис. 11.45. Градиентная раскраска

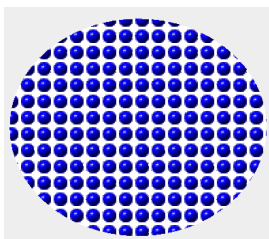


Рис. 11.46. Текстурная раскраска

Для создания объекта типа GradientPaint следует указать две точки изменения градиента и цвета, которые должны сменять друг друга в этих точках, как показано ниже.

```
g2.setPaint(new GradientPaint(p1, Color.RED,
                              p2, Color.YELLOW));
```

Постепенное изменение цвета осуществляется по прямой, соединяющей эти точки. Цвета остаются постоянными вдоль прямых, перпендикулярных к линии соединения. Для точек, находящихся за пределами соединительной линии, задаются цвета ее конечных точек соответственно.

Если же вызвать конструктор класса GradientPaint с параметром cyclic, принимающим логическое значение true, то изменение цвета будет происходить *циклически* за пределами конечных точек соединительной линии.

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2,
                              Color.YELLOW, true));
```

Для создания объекта типа TexturePaint следует указать объект типа BufferImage повторяющегося рисунка и *прямоугольник привязки*:

```
g2.setPaint(new TexturePaint(bufferedImage,
                             anchorRectangle));
```

Класс BufferedImage будет более подробно рассматриваться далее в этой главе, когда дойдет черед до обсуждения приемов обработки изображений. До тех пор достаточно сказать, что самый простой способ получить буферизированное изображение в виде объекта типа BufferedImage — ввести его из файла:

```
bufferedImage = ImageIO.read(new File("blue-ball.gif"));
```

Прямоугольник привязки повторяется бесконечное количество раз в направлениях, параллельных осям x и y, образуя текстуру в виде мозаики. Исходный рисунок масштабируется таким образом, чтобы вписаться в прямоугольник привязки, а затем повторяется в каждом элементе мозаики.

java.awt.Graphics2D 1.2

- **void setPaint(Paint *s*)**
  Устанавливает в качестве раскраски для данного графического контекста указанный объект, класс которого реализует интерфейс **Paint**.

java.awt.GradientPaint 1.2

- **GradientPaint(float *x1*, float *y1*, Color *color1*, float *x2*, float *y2*, Color *color2*)**
- **GradientPaint(float *x1*, float *y1*, Color *color1*, float *x2*, float *y2*, Color *color2*, boolean *cyclic*)**

java.awt.GradientPaint 1.2 *(окончание)*

- **GradientPaint(Point2D *p1*, Color *color1*, Point2D *p2*, Color *color2*)**
- **GradientPaint(Point2D *p1*, Color *color1*, Point2D *p2*, Color *color2*, boolean *cyclic*)**

  Создают объект градиентной раскраски. С его помощью фигура заполняется таким образом, чтобы начальная точка с координатами ***x1*** и ***y1*** была окрашена цветом ***color1***, конечная точка с координатами ***x2*** и ***y2*** — цветом ***color2***, а в промежутке между ними цвет изменялся линейно путем интерполяции. Цвета остаются постоянными вдоль прямой, перпендикулярной линии, соединяющей начальную и конечную точки. По умолчанию градиентная раскраска не выполняется циклически. Это означает, что точки, находящиеся за соответствующими пределами градиента, окрашиваются тем же самым цветом, что и начальная и конечная точки градиента. Если же градиентная раскраска выполняется *циклически*, то интерполяция цветов продолжается, возвращаясь сначала к цвету начальной точки, а затем непрерывно повторяясь в обоих направлениях.

java.awt.TexturePaint 1.2

- **TexturePaint(BufferedImage *texture*, Rectangle2D *anchor*)**

  Создает объект текстурной раскраски. Прямоугольник привязки определяет мозаичное заполнение раскрашиваемого участка. Он повторяется бесконечное число раз в направлении обеих осей координат *x* и *y*, масштабируя заданное изображение текстуры таким образом, чтобы оно заполняло каждый элемент мозаики.

11.3.6. Преобразование координат

Допустим, требуется нарисовать такой объект, как автомобиль, по известным исходным данным (высоте, расстоянию между осями и общей длине в метрах). Конечно, все координаты рисуемых частей автомобиля можно выразить в пикселях, подсчитав количество пикселей, приходящихся на метр. Но все это можно сделать намного проще, запросив соответствующий графический контекст и выполнив в нем преобразование координат следующим образом:

```
g2.scale(pixelsPerMeter, pixelsPerMeter);
// преобразовать в пиксели и нарисовать
// масштабированную соответственно линию:
g2.draw(new Line2D.Double(координаты_в_метрах));
```

Метод `scale()` из класса `Graphics2D` выполняет масштабное *преобразование координат* в заданном графическом контексте. При этом *пользовательские координаты* (единицы измерения, указанные пользователем) преобразуются в *аппаратные координаты* (пиксели). На рис. 11.47 приведен пример такого преобразования.

Преобразование координат очень удобно применять на практике, поскольку оно позволяет оперировать общепринятыми единицами измерения. Все хлопоты по их преобразованию в пиксели берет на себя заданный графический контекст.

Рис. 11.47. Пользовательские и аппаратные координаты

В языке Java предусмотрены четыре основных вида преобразований координат.

- **Масштабирование.** Увеличение или сокращение всех расстояний от фиксированной точки.
- **Вращение.** Поворот всех точек фигуры вокруг фиксированной точки.
- **Перемещение.** Передвижение всех точек фигуры на фиксированное расстояние.
- **Сдвиг.** Фиксация одной линии и перемещение всех параллельных ей линий фигуры на расстояние, пропорциональное расстоянию от данной линии до фиксированной.

На рис. 11.48 показаны результаты перечисленных выше видов преобразования, выполненных над единичным квадратом.

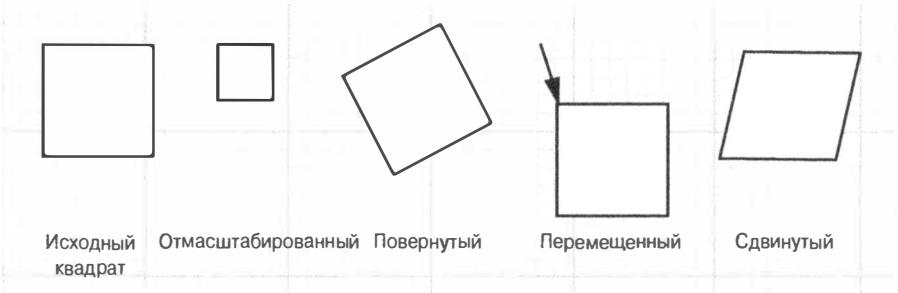


Рис. 11.48. Основные виды преобразования координат

Методы `scale()`, `rotate()`, `translate()` и `shear()` из класса `Graphics2D` реализуют перечисленные выше виды преобразования. Применяя эти методы в разных сочетаниях, можно выполнять *составные* преобразования, например, повернуть фигуру и удвоить ее размеры. Для этого достаточно выполнить операции вращения и масштабирования следующим образом:

```
g2.rotate(angle);
g2.scale(2, 2);
g2.draw(. . .);
```

В данном случае не важно, в какой именно последовательности будут выполняться преобразования. Но в целом порядок выполнения большинства преобразований *имеет* значение. Так, если требуется повернуть и сдвинуть изображение, сначала необходимо решить, какое именно преобразование выполнить первым, чтобы добиться желаемого результата. Преобразования в графическом контексте выполняются в обратном порядке по сравнению с тем, как они были указаны. Это означает, что преобразование, указанное последним, выполняется первым.

Допускается любое количество преобразований координат. Рассмотрим следующую последовательность преобразований:

```
g2.translate(x, y);
g2.rotate(a);
g2.translate(-x, -y);
```

Последнее преобразование (которое выполняется первым) перемещает всю фигуру таким образом, чтобы точка с координатами (x, y) совпала с точкой начала отсчета. Второе преобразование поворачивает фигуру на угол a вокруг точки начала отсчета. Последнее преобразование снова перемещает всю фигуру таким образом, чтобы точка с координатами (x, y) совпала с точкой начала отсчета. В итоге фигура поворачивается вокруг точки с координатами (x, y), как показано на рис. 11.49. Вращение фигуры вокруг точки, отличающейся от начала отсчета, выполняется довольно часто, поэтому для выполнения данной операции предусмотрен следующий метод:

```
g2.rotate(a, x, y);
```

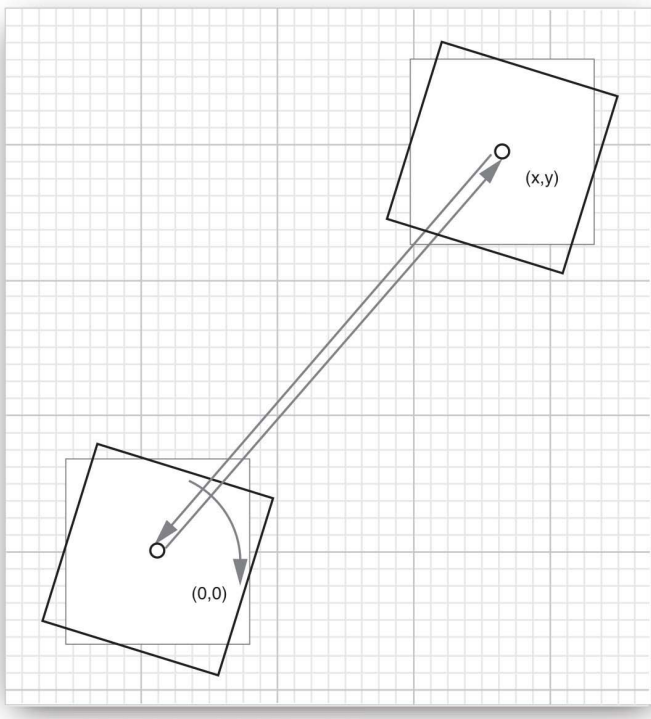


Рис. 11.49. Составные преобразования

Как известно из теории матриц, вращение, масштабирование, перемещение, сдвиг и различные их сочетания могут быть выражены в виде матриц преобразования следующим образом:

$$\begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Такие преобразования называются *аффинными*, а в прикладном интерфейсе Java 2D API они описываются с помощью класса `AffineTransform`. Объект аффинного преобразования можно непосредственно создать, если известны компоненты матрицы преобразования:

```
var t = new AffineTransform(a, b, c, d, e, f);
```

Для реализации отдельных типов преобразований предусмотрены также фабричные методы `getRotateInstance()`, `getScaleInstance()`, `getTranslateInstance()` и `getShearInstance()`, назначение которых можно легко определить по их названию. Например, при вызове метода

```
t = AffineTransform.getScaleInstance(2.0F, 0.5F);
```

возвращается преобразование, которое соответствует такой матрице:

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 0{,}5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Наконец, методы экземпляра `setToRotate()`, `setToScale()`, `setToTranslation()` и `setToShear()` служат для указания нового вида преобразования. Например:

```
// установить вращение на заданный угол:
t.setToRotation(angle);
```

Для замены текущего преобразования координат в графическом контексте аффинным преобразованием, представленным объектом типа `AffineTransform`, служит метод

```
g2.setTransform(t); // заменить текущее преобразование
```

Но на практике вызывать метод `setTransform()` все же не рекомендуется, поскольку он заменяет любое преобразование, которое может присутствовать в графическом контексте. Например, графический контекст для печати страницы с альбомной ориентацией уже содержит преобразование вращением на 90°, и поэтому при вызове метода `setTransform()` данное преобразование отменяется. В таком случае вместо метода `setTransform()` рекомендуется вызвать метод `transform()`, как показано ниже. Этот метод объединяет текущее преобразование с новым объектом типа `AffineTransform`, представляющим аффинное преобразование.

```
// составить текущее преобразование вместе с афинным:
g2.transform(t);
```

Если же требуется выполнить какое-нибудь преобразование временно, то сначала следует сохранить предыдущее преобразование, затем составить вместе

с ним новое преобразование и восстановить прежнее преобразование, как показано в приведенном ниже фрагменте кода.

```
// сохранить прежнее преобразование:
AffineTransform oldTransform = g2.getTransform();
g2.transform(t); // выполнить временное преобразование
рисовать в графическом контексте g2
// восстановить прежнее преобразование:
g2.setTransform(oldTransform);
```

java.awt.geom.AffineTransform 1.2

- **double *f*)**
- **AffineTransform(float *a*, float *b*, float *c*, float *d*, float *e*, float *f*)**
  Конструируют аффинное преобразование по приведенной ниже матрице.

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

- **AffineTransform(double[] *m*)**
- **AffineTransform(float[] *m*)**
  Конструируют аффинное преобразование по приведенной ниже матрице.

$$\begin{bmatrix} m[0] & m[2] & m[4] \\ m[1] & m[3] & m[5] \\ 0 & 0 & 1 \end{bmatrix}$$

- **static AffineTransform getRotateInstance(double *a*)**
  Задает преобразование вращением против часовой стрелки вокруг точки начала отсчета на угол ***a***, указываемый в радианах. Ниже приведена матрица такого преобразования.

$$\begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  Если значение параметра ***a*** находится в пределах от **0** до **π/2**, то вращение происходит в направлении от положительной оси *x* до положительной оси *y*.
- **static AffineTransform getRotateInstance(double *a*, double *x*, double *y*)**
  Задает преобразование вращением против часовой стрелки вокруг точки с координатами [***x***, ***y***] на угол ***a***, указываемый в радианах.
- **static AffineTransform getScaleInstance(double *sx*, double *sy*)**
  Задает преобразование масштабированием с масштабными коэффициентами ***sx*** и ***sy*** по осям *x* и *y* соответственно. Ниже приведена матрица такого преобразования.

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

java.awt.geom.AffineTransform 1.2 *(окончание)*

- **static AffineTransform getShearInstance (double *shx*, double *shy*)**

  Задает преобразование сдвигом с коэффициентами ***shx*** и ***shy*** по осям *x* и *y*. Ниже приведена матрица такого преобразования.**AffineTransform(double *a*, double *b*, double *c*, double *d*, double *e*, static AffineTransform getTranslateInstance(double *tx*, double *ty***)

  Задает преобразование перемещением на расстояния ***tx*** и ***ty*** по осям *x* и *y*. Ниже приведена матрица такого преобразования.

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

- **void setToRotation(double *a*)**
- **void setToRotation(double *a*, double *x*, double *y*)**
- **void setToScale(double *sx*, double *sy*)**
- **void setToShear(double *sx*, double *sy*)**
- **void setToTranslation(double *tx*, double *ty*)**

  Устанавливают аффинное преобразование в соответствии с указанными параметрами. Эти параметры интерпретируются таким же образом, как и в упомянутых выше методах типа **get*Xxx*Instance()** для основных преобразований.

java.awt.Graphics2D 1.2

- **void setTransform(AffineTransform *t*)**

  Заменяет существующее преобразование координат в данном графическом контексте указанным аффинным преобразованием ***t***.

- **void transform(AffineTransform *t*)**

  Составляет преобразование координат, существующее в данном графическом контексте, вместе с аффинным преобразованием ***t***.

- **void rotate(double *a*)**
- **void rotate(double *a*, double *x*, double *y*)**
- **void scale(double *sx*, double *sy*)**
- **void shear(double *sx*, double *sy*)**
- **void translate(double *tx*, double *ty*)**

  Составляют преобразование координат, существующее в данном графическом контексте, вместе с основным преобразованием по указанным параметрам. Эти параметры интерпретируются таким же образом, как и в упомянутых выше методах типа **get*Xxx*Instance()** для основных преобразований.

11.3.7. Отсечение

Для выполнения всех графических операций только внутри ограниченного участка в графическом контексте предусмотрена *фигура отсечения*:

```
g2.setClip(clipShape); // можно, но лучше все же
    // воспользоваться приведенным ниже методом:
g2.draw(shape); // рисовать только внутри фигуры отсечения
```

Но на практике вызывать метод `setClip()` не рекомендуется, поскольку он заменяет все существующие фигуры отсечения в данном графическом контексте. Как будет показано далее, графический контекст для печати уже содержит прямоугольник отсечения, который позволяет избежать появления данных на полях страницы. В этом случае вместо метода `setClip()` лучше вызвать метод `clip()` следующим образом:

```
g2.clip(clipShape); // лучше вызвать именно этот метод
```

Метод `clip()` образует пересечение существующей фигуры отсечения с указанной фигурой. Если же фигуру отсечения требуется применить временно, то сначала следует сохранить прежнюю фигуру, затем добавить новую и восстановить прежнюю. Ниже приведен характерный тому пример.

```
Shape oldClip = g2.getClip(); // сохранить прежнее отсечение
g2.clip(clipShape); // произвести временное отсечение
рисовать в графическом контексте g2
g2.setClip(oldClip); // восстановить прежнее отсечение
```

На рис. 11.50 возможности отсечения демонстрируются на примере рисования довольно непростого штрихового рисунка, который отсекается сложной фигурой, а именно контуром ряда символов.

Рис. 11.50. Отсечение штрихового рисунка фигурами букв

Для получения контуров символов требуется *контекст воспроизведения шрифтов*. С этой целью сначала вызывается метод `getFontRenderContext()` из класса `Graphics2D`:

```
FontRenderContext context = g2.getFontRenderContext();
```

Затем создается объект расположения текста типа `TextLayout`, для чего используется символьная строка, шрифт и контекст воспроизведения шрифтов:

```
var layout = new TextLayout("Hello", font, context);
```

Объект расположения текста описывает последовательность символов, расположение и оформление которых определяется выбранным контекстом

воспроизведения шрифтов. Как известно, одни и те же символы могут по-разному выглядеть на экране и на печатной странице.

Но важнее другое: метод `getOutline()` возвращает объект типа `Shape`, описывающий фигуру контура символов в расположении текста. Эта фигура начинается в точке начала отсчета с координатами (0,0). Но если такое расположение не подходит, то при вызове метода `getOutline()` задается аффинное преобразование, позволяющее указать, в какой именно точке должен появиться контур:

```
AffineTransform transform =
      AffineTransform.getTranslateInstance(0, 100);
Shape outline = layout.getOutline(transform);
```

Затем контур присоединяется к фигуре отсечения следующим образом:

```
var clipShape = new GeneralPath();
clipShape.append(outline, false);
```

Наконец, устанавливается фигура отсечения и рисуются линии штриховки, как показано ниже. В итоге линии появляются только внутри символов.

```
g2.setClip(clipShape);
var p = new Point2D.Double(0, 0);
for (int i = 0; i < NLINES; i++)
{
  double x = . . .;
  double y = . . .;
  var q = new Point2D.Double(x, y);
  // отсечь линии штриховки:
  g2.draw(new Line2D.Double(p, q));
}
```

java.awt.Graphics 1.0

- **void setClip(Shape *s*)** 1.2
  Задает фигуру ***s*** в качестве текущей фигуры отсечения.
- **Shape getClip()** 1.2
  Возвращает текущую фигуру отсечения.

java.awt.Graphics2D 1.2

- **void clip(Shape *s*)**
  Образует пересечение текущей фигуры отсечения с заданной фигурой ***s***.
- **FontRenderContext getFontRenderContext()**
  Возвращает контекст воспроизведения шрифтов, требующийся для создания объекта типа **TextLayout**.

java.awt.font.TextLayout 1.2

- **TextLayout(String *s*, Font *f*, FontRenderContext *context*)**
  Создает объект **TextLayout**, исходя из заданной символьной строки, шрифта и контекста воспроизведения шрифтов.
- **float getAdvance()**
  Возвращает ширину данного расположения текста.
- **float getAscent()**
- **float getDescent()**
  Возвращают высоту данного расположения текста относительно базовой линии.
- **float getLeading()**
  Возвращает расстояние между соседними строками для шрифта, применяемого в данном расположении текста.

11.3.8. Прозрачность и композиция

В стандартной цветовой модели RGB каждый цвет описывается по трем его основным составляющим: красной, зеленой и синей. Но иногда отдельные участки изображения удобно сделать *прозрачными* или частично прозрачными. При наложении рисунка на уже существующее изображение прозрачные пиксели не закрывают находящиеся под ними пиксели, а частично прозрачные пиксели смешиваются с нижележащими пикселями. На рис. 11.51 приведен результат наложения частично прозрачного прямоугольника на уже имеющееся изображение. Обратите внимание на то, что детали изображения, находящиеся под прямоугольником, по-прежнему видны.

Рис. 11.51. Наложение частично прозрачного прямоугольника на изображение

В прикладном интерфейсе Java 2D API прозрачность описывается с помощью *альфа-канала*. Каждый пиксель, кроме красной, зеленой и синей составляющей цвета, имеет значение прозрачности в альфа-канале, изменяющееся в пределах от 0 (полностью прозрачен) до 1 (совершенно непрозрачен). Например, прямоугольник на рис. 11.51 заполнен бледно-желтым цветом с прозрачностью 50% следующим образом:

```
new Color(0.7F, 0.7F, 0.0F, 0.5F);
```

Теперь рассмотрим пример наложения двух фигур. Для этого требуется смешать или составить цвета и значения прозрачности в альфа-канале исходных

и целевых пикселей. Исследователи Портер (Porter) и Дафф (Duff) в области компьютерной графики сформулировали двенадцать возможных *правил композиции*, которые реализованы в прикладном интерфейсе Java 2D API. Однако только два из них имеют практическое применение. Если эти правила покажутся слишком сложными, то вместо них рекомендуется использовать простое правило `SRC_OVER`, которое применяется по умолчанию к объектам типа `Graphics2D` и дает интуитивно понятные результаты.

Рассмотрим вкратце теоретические основы правил композиции. Допустим, *исходный пиксель* имеет значение прозрачности в альфа-канале a_S, а *целевой пиксель* — значение прозрачности a_D. На рис. 11.52 показана диаграмма, схематически поясняющая правило композиции для этих значений.

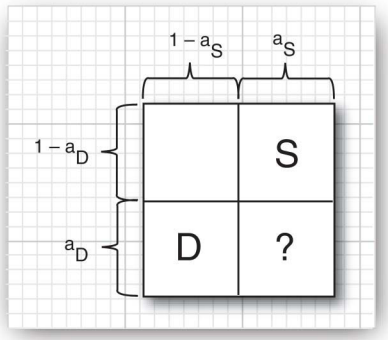


Рис. 11.52. Схематическое представление правила композиции

Портер и Дафф считают, что значение прозрачности в альфа-канале выражает вероятность использования цвета пикселя при объединении изображений. Для исходного пикселя первоначальный цвет будет использоваться с вероятностью a_S и не использоваться с вероятностью $1 - a_S$. Это же справедливо и для целевого пикселя. Допустим, при составлении цветов эти вероятности независимы. На рис. 11.52 показаны четыре возможные ситуации. Если требуется использовать преимущественно цвет исходного, а не целевого пикселя (эта ситуация обозначена буквой `S`), то вероятность такого события равна $a_S \cdot (1 - a_D)$. Аналогично вычисляется вероятность $a_D \cdot (1 - a_S)$ преимущественного использования цвета целевого, а не исходного пикселя (эта ситуация обозначена буквой `D`). Что же делать, если в исходном и целевом изображениях преимущественно выбирается свой цвет? Именно в этой ситуации используются правила композиции Портера–Даффа. Если предпочтение отдается исходному цвету, в таком случае правый нижний угол диаграммы помечается буквой `S`, а само правило композиции называется `SRC_OVER`. По этому правилу исходный и целевой цвета сочетаются с весом a_S и $a_D \cdot (1 - a_S)$ соответственно.

Визуальный эффект применения этого правила композиции состоит в том, что при смешении цветов исходного и целевого пикселей приоритет отдается цвету исходного пикселя. В частности, если $a_S = 1$, то цвет целевого пикселя вообще не принимается во внимание. Если $a_S = 0$, то исходный пиксель становится совершенно прозрачным, тогда как цвет целевого пикселя не изменяется.

В зависимости от того, как расставлены буквы на диаграмме, можно сформировать и другие правила композиции. Все правила композиции, поддерживаемые в прикладном интерфейсе Java 2D API, перечислены в табл. 11.3 и схематически показаны на рис. 11.53. Изображения на этом рисунке представляют результаты применения правил композиции к прямоугольному участку исходного изображения со степенью прозрачности `0,75` в альфа-канале и эллиптическому участку целевого изображения со степенью прозрачности `1,0` в альфа-канале.

Как можно заметить, большинство этих правил вряд ли применяются на практике. Так, например, правило композиции `DST_IN` представляет собой крайний случай, когда во внимание совсем не принимается цвет исходного пикселя, тогда как его альфа-канал используется для изменения целевого пикселя. В отличие от него, правило композиции `SRC` может оказаться удобным, потому что оно предписывает использовать исходный цвет, исключая смешение с целевым цветом. Более подробно о правилах Портера–Даффа можно прочитать в упоминавшейся ранее книге *Computer Graphics: Principles and Practice, Third Edititon* Джеймса Фоли, Андриса ван Дама, Стивена Фэйнера и др.

Для создания объекта правила композиции, класс которого реализует интерфейс `Composite`, служит метод `setComposite()` из класса `Graphics2D`. В состав прикладного интерфейса Java 2D API входит лишь один такой класс — `AlphaComposite`. В этом классе реализованы все правила Портера–Даффа, представленные на рис. 11.53.

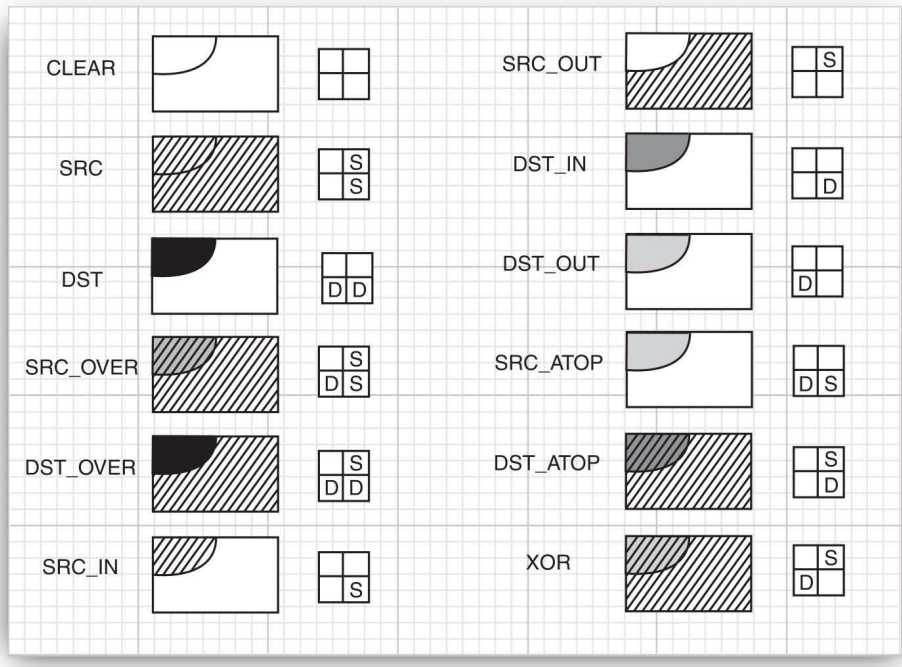


Рис. 11.53. Правила композиции Портера–Даффа

Таблица 11.3. Правила композиции Портера–Даффа

Правило	Описание
CLEAR	Исходные пиксели очищают целевые
SRC	Исходные пиксели перезаписывают целевые и пустые пиксели
DST	Исходные пиксели не оказывают никакого влияния на целевые пиксели
SRC_OVER	Исходные пиксели смешиваются с целевыми и перезаписывают пустые пиксели
DST_OVER	Исходные пиксели не оказывают никакого влияния на целевые пиксели и перезаписывают пустые пиксели
SRC_IN	Исходные пиксели перезаписывают целевые пиксели
SRC_OUT	Исходные пиксели очищают целевые пиксели и перезаписывают пустые пиксели
DST_IN	Прозрачность в альфа-канале исходного изображения видоизменяет целевое изображение
DST_OUT	Дополнение до прозрачности в альфа-канале исходного изображения видоизменяет целевое изображение
SRC_ATOP	Исходные пиксели смешиваются с целевыми
DST_ATOP	Прозрачность в альфа-канале исходного изображения видоизменяет целевое изображение. Исходные пиксели перезаписывают пустые пиксели
XOR	Дополнение до прозрачности в альфа-канале исходного изображения видоизменяет целевое изображение. Исходные пиксели перезаписывают пустые пиксели

Для создания экземпляра правила типа `AlphaComposite` служит фабричный метод `getInstance()` из класса `AlphaComposite`. В качестве параметров этого метода указываются правило композиции и значение прозрачности в альфа-канале для пикселей исходного изображения, как показано в приведенном ниже фрагменте кода.

```
int rule = AlphaComposite.SRC_OVER;
float alpha = 0.5f;
g2.setComposite(AlphaComposite.getInstance(rule, alpha));
g2.setPaint(Color.blue);
g2.fill(rectangle);
```

В этом фрагменте кода прямоугольник заполняется синим цветом со степенью прозрачности `0,5` в альфа-канале. В соответствии с заданным правилом композиции `SRC_OVER` этот прямоугольник прозрачно накладывается на уже существующее изображение.

В примере программы из листинга 11.17 предоставляется возможность исследовать правила композиции. Для этого достаточно выбрать конкретное правило из комбинированного списка и установить ползунковым регулятором степень прозрачности в альфа-канале для объекта типа `AlphaComposite`.

В нижней части рабочего окна данной программы приводится краткое описание каждого правила композиции. Обратите внимание на то, что это описание автоматически составляется по диаграммам правил композиции. Например, строка **"DS"** во втором ряду диаграммы приводит к появлению в описании правила композиции строки "blends with destination" (смешивается с цветом целевого изображения).

Но эта программа совсем не гарантирует, что используемый графический контекст, соответствующий экрану, имеет альфа-канал. Когда пиксели накладываются на целевое изображение без альфа-канала, цвета пикселей умножаются на степень прозрачности в альфа-канале, а затем эта степень отбрасывается. В ряде правил композиции Портера–Даффа используются степени прозрачности в альфа-канале целевого изображения, и это указывает на важную роль альфа-канала. Поэтому для составления изображений (в данном случае — фигур) используется буферизированное изображение с цветовой моделью ARGB. После составления результирующее изображение выводится на экран, как показано в приведенном ниже фрагменте кода.

```
var image = new BufferedImage(getWidth(), getHeight(),
                              BufferedImage.TYPE_INT_ARGB);
Graphics2D gImage = image.createGraphics();
// а теперь рисовать на изображении gImage
g2.drawImage(image, null, 0, 0);
```

В листингах 11.17 и 11.18 представлен исходный код классов фрейма и компонента. Класс `Rule` из листинга 11.19 предоставляет краткое описание каждого правила композиции, как показано на рис. 11.54. После запуска программы на выполнение переместите ползунок регулятора слева направо, чтобы посмотреть результат применения выбранного правила композиции для наложения фигур при разных степенях прозрачности в альфа-канале. Обратите особое внимание на то, что правила `DST_IN` и `DST_OUT` отличаются лишь направлением изменения цвета целевого изображения (!) при изменении степени прозрачности в альфа-канале исходного изображения.

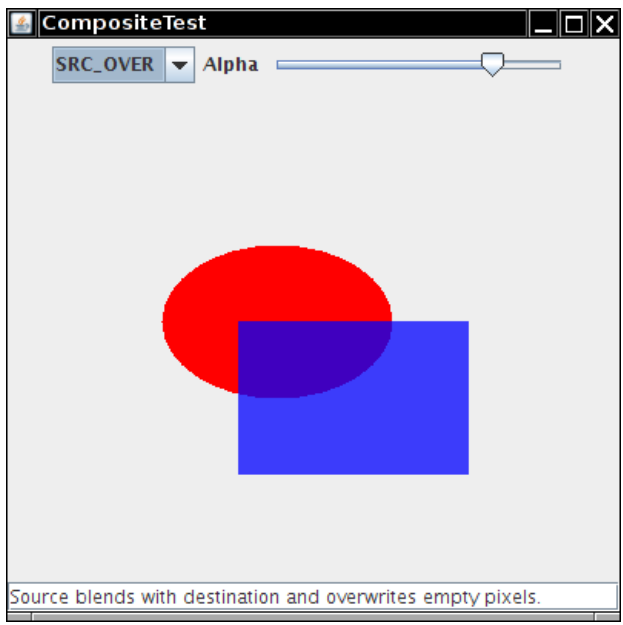


Рис. 11.54. Рабочее окно программы `CompositeTest`

Листинг 11.17. Исходный код из файла **composite/CompositeTestFrame.java**

```
1  package composite;
2
3  import java.awt.*;
4
5  import javax.swing.*;
6
7  /**
8   * Этот фрейм содержит комбинированный список для
9   * выбора правил, композиции, ползунок для изменения
10  * прозрачности в альфа-канале исходного изображения,
11  * а также компонент для отображения результатов
12  * составления изображений
13  */
14 class CompositeTestFrame extends JFrame
15 {
16   private static final int DEFAULT_WIDTH = 400;
17   private static final int DEFAULT_HEIGHT = 400;
18
19   private CompositeComponent canvas;
20   private JComboBox<Rule> ruleCombo;
21   private JSlider alphaSlider;
22   private JTextField explanation;
23
24   public CompositeTestFrame()
25   {
26     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
27
28     canvas = new CompositeComponent();
29     add(canvas, BorderLayout.CENTER);
30
31     ruleCombo = new JComboBox<>(new Rule[]
32                 { new Rule("CLEAR", " ", " "),
33                   new Rule("SRC", " S", " S"),
34                   new Rule("DST", " ", "DD"),
35                   new Rule("SRC_OVER", " S", "DS"),
36                   new Rule("DST_OVER", " S", "DD"),
37                   new Rule("SRC_IN", " ", " S"),
38                   new Rule("SRC_OUT", " S", " "),
39                   new Rule("DST_IN", " ", " D"),
40                   new Rule("DST_OUT", " ", "D "),
41                   new Rule("SRC_ATOP", " ", "DS"),
42                   new Rule("DST_ATOP", " S", " D"),
43                   new Rule("XOR", " S", "D "), });
44     ruleCombo.addActionListener(event ->
45       {
46         var r = (Rule) ruleCombo.getSelectedItem();
47         canvas.setRule(r.getValue());
48         explanation.setText(r.getExplanation());
49       });
50
51     alphaSlider = new JSlider(0, 100, 75);
52     alphaSlider.addChangeListener(event ->
53          canvas.setAlpha(alphaSlider.getValue()));
```

```
54      var panel = new JPanel();
55      panel.add(ruleCombo);
56      panel.add(new JLabel("Alpha"));
57      panel.add(alphaSlider);
58      add(panel, BorderLayout.NORTH);
59
60      explanation = new JTextField();
61      add(explanation, BorderLayout.SOUTH);
62
63      canvas.setAlpha(alphaSlider.getValue());
64      Rule r = ruleCombo.getItemAt(
65      ruleCombo.getSelectedIndex());
66      canvas.setRule(r.getValue());
67      explanation.setText(r.getExplanation());
68    }
69 }
```

Листинг 11.18. Исходный код из файла **composite/CompositeComponent.java**

```
1  package composite;
2
3  import java.awt.*;
4  import java.awt.geom.*;
5  import java.awt.image.*;
6  import javax.swing.*;
7
8  /**
9   * Этот компонент рисует две формы,
10  * составленные по правилу композиции
11  */
12 class CompositeComponent extends JComponent
13 {
14   private int rule;
15   private Shape shape1;
16   private Shape shape2;
17   private float alpha;
18
19   public CompositeComponent()
20   {
21     shape1 = new Ellipse2D.Double(100, 100, 150, 100);
22     shape2 = new Rectangle2D.Double(150, 150, 150, 100);
23   }
24
25   public void paintComponent(Graphics g)
26   {
27     var g2 = (Graphics2D) g;
28
29     var image = new BufferedImage(getWidth(),
30         getHeight(), BufferedImage.TYPE_INT_ARGB);
31     Graphics2D gImage = image.createGraphics();
32     gImage.setPaint(Color.red);
33     gImage.fill(shape1);
34     AlphaComposite composite =
35          AlphaComposite.getInstance(rule, alpha);
36     gImage.setComposite(composite);
```

```
37      gImage.setPaint(Color.blue);
38      gImage.fill(shape2);
39      g2.drawImage(image, null, 0, 0);
40    }
41
42    /**
43     * Устанавливает правило композиции
44     * @param r Правило композиции (в виде константы
45     *          из класса AlphaComposite)
46    */
47    public void setRule(int r)
48    {
49      rule = r;
50      repaint();
51    }
52
53    /**
54     * Устанавливает значение прозрачности в
55     * альфа-канале исходного изображения
56     * @param a Значение прозрачности в пределах
57     * от 0 до 100
58    */
59    public void setAlpha(int a)
60    {
61      alpha = (float) a / 100.0F;
62      repaint();
63    }
64  }
```

Листинг 11.19. Исходный код из файла **composite/Rule.java**

```
1   package composite;
2
3   import java.awt.*;
4
5   /**
6    * Этот класс описывает правило композиции Портера-Даффа
7   */
8   class Rule
9   {
10    private String name;
11    private String porterDuff1;
12    private String porterDuff2;
13
14    /**
15     * Составляет правило композиции Портера-Даффа
16     * @param n Название правила композиции
17     * @param pd1 Первый ряд квадратов в правиле
18     *            композиции Портера-Дафа
19     * @param pd2 Второй ряд в квадратов правиле
21     *            композиции Портера-Дафа
22     */
23    public Rule(String n, String pd1, String pd2)
24    {
25      name = n;
```

```
26      porterDuff1 = pd1;
27      porterDuff2 = pd2;
27    }
28
29    /**
30     * Получает объяснение композиции по данному правилу
31     * @return Объяснение композиции по данному правилу
32     */
33    public String getExplanation()
34    {
35      var r = new StringBuilder("Source ");
36      if (porterDuff2.equals(" "))
37        r.append("clears");
38      if (porterDuff2.equals(" S"))
39        r.append("overwrites");
40      if (porterDuff2.equals("DS"))
41        r.append("blends with");
42      if (porterDuff2.equals(" D"))
43        r.append("alpha modifies");
44      if (porterDuff2.equals("D "))
45        r.append("alpha complement modifies");
46      if (porterDuff2.equals("DD"))
47        r.append("does not affect");
48      r.append(" destination");
49      if (porterDuff1.equals(" S"))
50        r.append(" and overwrites empty pixels");
51      r.append(".");
52      return r.toString();
53    }
54
55    public String toString()
56    {
57      return name;
58    }
59
60    /**
61     * Получает значение по данному правилу
62     * в классе AlphaComposite
63     * @return Значение константы из класса
64     *         AlphaComposite или значение -1, если
65     *         соответствующая константа отсутствует
66     */
67    public int getValue()
68    {
69      try
70      {
71        return (Integer) AlphaComposite.class
72                         .getField(name).get(null);
73      }
74      catch (Exception e)
75      {
76        return -1;
77      }
78    }
79  }
```

java.awt.Graphics2D 1.2

- **void setComposite(Composite *s*)**

  Устанавливает указанный объект, класс которого реализует интерфейс **Composite**, в качестве правила композиции для данного графического контекста.

java.awt.AlphaComposite 1.2

- **static AlphaComposite getInstance(int *rule*)**
- **static AlphaComposite getInstance(int *rule*, float *sourceAlpha*)**

  Создают объект композиции на основе заданного правила и значения в альфа-канале. Параметр ***rule***, задающий правило композиции, может принимать одно из следующих значений: **CLEAR**, **DST**, **SRC**, **SRC_OVER**, **DST_OVER**, **SRC_ATOP**, **SRC_IN**, **SRC_OUT**, **DST_ATOP**, **DST_IN**, **DST_OUT**, **XOR**.

11.4. Растровые изображения

Применяя прикладной интерфейс Java2D API, можно создавать рисунки, состоящие из линий, кривых и участков. Этот прикладной интерфейс служит для построения векторной графики, поскольку для этой цели требуется указывать математические свойства фигур. Но для обработки изображений, состоящих из отдельных пикселей, требуется оперировать растровыми данными цвета. В последующих разделах поясняется, каким образом организуется обработка растровых изображений в Java.

11.4.1. Чтение и запись изображений

В пакете `javax.imageio` содержатся готовые средства для чтения и записи файлов изображений в ряде наиболее употребительных форматов, а также библиотека для чтения и записи файлов других форматов. В частности, поддерживаются форматы GIF, JPEG, PNG, BMP (растровый формат для Windows) и WBMP (Wireless Bitmap — растровый формат для беспроводных сетей).

Основные функциональные возможности, доступные в библиотеке для чтения и записи изображений, чрезвычайно просты. Так, для загрузки изображения из файла применяется статический метод `read()` из класса `ImageIO`:

```
File f = . . .;
BufferedImage image = ImageIO.read(f);
```

Класс `ImageIO` выбирает соответствующее средство чтения, исходя из типа файла. Он проверяет расширение файла и соответствующее значение в заголовке файла. Если для чтения данного файла нельзя найти подходящее средство или оно не в состоянии расшифровать содержимое файла, то статический метод `read()` из этого класса возвращает пустое значение `null`.

Так же просто осуществляется запись изображения в файл, как показано в приведенном ниже фрагменте кода. В данном случае форматирующая строка

в переменной `format` определяет формат изображения (например, JPEG или PNG), а класс `ImageIO` выбирает соответствующее средство записи и сохраняет изображение в файле.

```
File f = . . .;
String format = . . .;
ImageIO.write(image, format, f);
```

11.4.1.1. Получение средств чтения и записи изображений по типам файлов

Для выполнения расширенных операций записи и чтения изображений, которые выходят за пределы простого использования статических методов `read()` и `write()` из класса `ImageIO`, необходимо прежде всего получить объекты типа `ImageReader` и `ImageWriter` соответственно. В классе `ImageIO` перечисляются средства чтения и записи, отвечающие одному из следующих условий.

- Формат изображения, например JPEG.
- Расширение файла, например **`jpg`**.
- Тип MIME, например `image/jpeg`.

НА ЗАМЕТКУ! Стандарт MIME (Multipurpose Internet Mail Extensions — многоцелевые расширения почты в Интернете) определяет общие форматы данных (например, **`image/jpeg`** или **`application/pdf`**).

Например, с помощью приведенного ниже фрагмента кода можно получить средство чтения файлов изображений формата JPEG. Методы `getImageReaderBySuffix()` и `getImageReaderByMIMEType()` возвращают средства чтения файлов изображений с указанным расширением или типом MIME.

```
ImageReader reader = null;
Iterator<ImageReader> iter =
         ImageIO.getImageReadersByFormatName("JPEG");
if (iter.hasNext()) reader = iter.next();
```

Класс `ImageIO` позволяет обнаружить несколько средств чтения, каждое из которых способно обработать файлы изображений конкретного типа. В этом случае средство чтения выбирается исходя из более подробных сведений, которые можно получить с помощью интерфейса *поставщика услуг* следующим образом:

```
ImageReaderSpi spi = reader.getOriginatingProvider();
```

Затем можно получить название поставщика и номер версии:

```
String vendor = spi.getVendor();
String version = spi.getVersion();
```

Возможно, эти сведения помогут сделать выбор подходящего средства чтения или хотя бы составить список доступных средств, чтобы пользователи могли выбрать наиболее подходящее из них по своему усмотрению. Но для начала будем считать, что для чтения файлов изображений подходит первое же перечисленное средство.

В примере программы из листинга 11.20 требуется найти расширения файлов для всех доступных средств чтения, чтобы использовать их в фильтре файлов. Для

этой цели следует вызвать статический метод `ImageIO.getReaderFileSuffixes()`, как показано ниже.

```
String[] extensions = ImageIO.getWriterFileSuffixes();
chooser.setFileFilter(new FileNameExtensionFilter(
                      "Image files", extensions));
```

Что же касается сохранения изображений в файлах, то для этого потребуется больше усилий. Хотелось бы, конечно, предоставить пользователю меню со всеми поддерживаемыми форматами изображений. Но, к сожалению, метод `getWriterFormaNames()` из класса `IOImage` для этой цели не подходит, потому что он возвращает не совсем обычный перечень с лишними вариантами названий форматов, например, следующий:

```
jpg, BMP, bmp, JPG, jpeg, wbmp, png, JPEG, PNG, WBMP, GIF, gif
```

Но это не совсем то, что должно отображаться в предполагаемом меню. Ведь это должен быть перечень только предпочитаемых названий форматов. Поэтому для этой цели используется вспомогательный метод `getWriterFormats()` (см. листинг 11.20). Сначала в этом методе отыскивается первое средство записи, ассоциируемое с названием каждого формата, а затем у него запрашиваются имеющиеся названия форматов в надежде, что первым будет перечислен наиболее подходящий формат. Для средства записи изображений в файлы формата JPEG такой подход вполне пригоден, поскольку первым это средство, естественно, перечислит формат JPEG. (А вот средство записи изображений в файлы формата PNG перечислит первым не название формата PNG, а его строчный эквивалент `png`. Можно надеяться, что в ближайшем будущем этот недостаток будет устранен, а до тех пор все строчные буквы в названии форматов придется преобразовать в прописные.) После выбора названия предпочтительного формата все его альтернативные названия просто удаляются из исходного набора, и так происходит до тех пор, пока не будут обработаны все названия форматов.

11.4.1.2. Чтение и запись файлов с несколькими изображениями

Некоторые файлы, например анимационного формата GIF, могут содержать несколько изображений. Но статический метод `read()` из класса `ImageIO` позволяет прочитать только одно из них. Для чтения нескольких изображений необходимо преобразовать сначала источник входных данных (например, поток ввода или файл) в объект потока ввода изображений, относящийся к типу `ImageInputStream`, следующим образом:

```
InputStream in = . . .;
ImageInputStream imageIn =
     ImageIO.createImageInputStream(in);
```

Затем этот объект следует присоединить к средству чтения, вызвав следующий метод:

```
reader.setInput(imageIn, true);
```

Второй параметр этого метода принимает логическое значение `true`, а это означает, что поток ввода находится в режиме просмотра данных только в прямом направлении. Логическое значение `false` этого параметра допускает произвольный

доступ. В этом случае данные, читаемые из потока ввода, буферизуются, или вводятся в режиме произвольного доступа к файлу. Произвольный доступ требуется только для определенных операций. Например, для подсчета количества изображений в файле анимационного формата GIF нужно прочитать весь этот файл. Если затем потребуется извлечь какое-нибудь изображение, то придется снова прочитать все введенные данные.

Это имеет значение только для чтения из потока ввода, если файл содержит несколько изображений, а формат изображения в его заголовке не предоставляет нужных сведений, например, о количестве изображений. Для чтения изображений непосредственно из файла можно воспользоваться приведенным ниже фрагментом кода.

```
File f = . . .;
ImageInputStream imageIn =
     ImageIO.createImageInputStream(f);
reader.setInput(imageIn);
```

Присоединив средство чтения к потоку ввода изображений, можно приступать к вводу из этого потока, вызвав приведенный ниже метод, где `index` — это индекс изображений, начиная с нуля.

```
BufferedImage image = reader.read(index);
```

Если поток ввода находится в режиме просмотра данных только в прямом направлении, то изображения должны считываться до тех пор, пока метод `read()` не сгенерирует исключение типа `IndexOutOgBoundsException`. В противном случае следует вызвать метод `getNumImages()`, как показано ниже.

```
int n = reader.getNumImages(true);
```

Параметр этого метода принимает логическое значение `true`, а это означает, что при вводе разрешается поиск и подсчет количества изображений. Следует, однако, иметь в виду, что метод `getNumImages()` генерирует исключение типа `IllegalStateException`, если поток ввода находится в режиме просмотра данных только в прямом направлении. Данный метод возвращает значение `-1`, если он не в состоянии определить количество изображений без поиска. В этом случае изображения придется считывать до тех пор, пока не возникнет исключение типа `IndexOutOfBoundsException`.

Некоторые файлы могут содержать миниатюрные виды изображений для их предварительного просмотра. Количество миниатюрных видов изображений можно выяснить, сделав следующий вызов:

```
int count = reader.getNumThumbnails(index);
```

Извлечь конкретный миниатюрный вид по его индексу можно следующим образом:

```
BufferedImage thumbnail =
        reader.getThumbnail(index, thumbnailIndex);
```

Иногда размеры изображения требуется получить еще до его извлечения из файла. Это особенно важно, когда речь идет о крупном изображении или его

передаче по низкоскоростному сетевому соединению. Для получения размеров изображения по указанному индексу вызываются следующие методы:

```
int width = reader.getWidth(index);
int height = reader.getHeight(index);
```

Для записи нескольких изображений в файл необходимо создать сначала объект типа `ImageWriter`, а затем с помощью класса `IOImage` перечислить все средства, способные записывать изображения в конкретном формате:

```
String format = . . .;
ImageWriter writer = null;
Iterator<ImageWriter> iter =
         ImageIO.getImageWritersByFormatName(format);
if (iter.hasNext()) writer = iter.next();
```

Затем поток вывода или файл следует преобразовать в поток вывода изображений (объект типа `ImageOutputStream`) и присоединить его к средству записи следующим образом:

```
File f = . . .;
ImageOutputStream imageOut =
     ImageIO.createImageOutputStream(f);
writer.setOutput(imageOut);
```

Каждое изображение следует заключить в оболочку объекта типа `IIOImage`, как показано ниже. Дополнительно можно предоставить список миниатюрных видов и метаданные (например, алгоритм сжатия данных и цветовую информацию). В рассматриваемом здесь примере вместо списка миниатюрных видов и метаданных указываются пустые значения `null`. За дополнительной справкой по данному вопросу обращайтесь к документации на соответствующий прикладной интерфейс API.

```
var iioImage = new IIOImage(images[i], null, null);
```

Для записи *первого* изображения вызывается метод `write()`:

```
writer.write(new IIOImage(images[0], null, null));
```

Для записи всех последующих изображений служит код из приведенного ниже примера. В качестве третьего параметра при вызове метода `write()` может быть указан объект типа `ImageWriteParam`, предоставляющий такие подробности, как мозаичное расположение и алгоритм сжатия данных. В данном примере в качестве третьего параметра указано пустое значение `null`.

```
if (writer.canInsertImage(i))
  writer.writeInsert(i, iioImage, null);
```

Не все форматы допускают сохранение нескольких изображений в файлах. В этом случае метод `canInsertImage()` возвращает логическое значение `false` по условию `i > 0`, и тогда в файле сохраняется только одно изображение. В примере программы из листинга 11.20 изображения загружаются и сохраняются в файлах тех форматов, для которых в библиотеке Java предусмотрены средства чтения и записи. Эта программа позволяет отображать сразу несколько изображений, но не их миниатюрные виды, как показано на рис. 11.55.

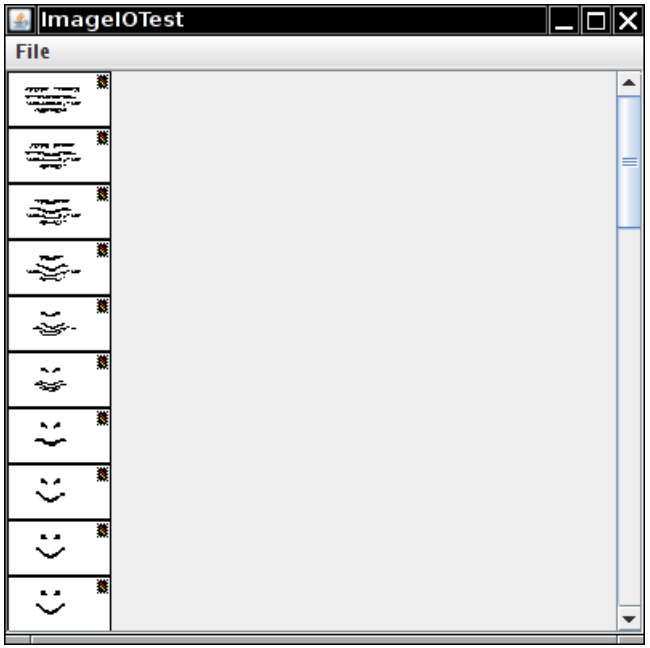


Рис. 11.55. Анимационная последовательность изображений в формате GIF

Листинг 11.20. Исходный код из файла **imageIO/ImageIOFrame.java**

```
1   package imageIO;
2
3   import java.awt.image.*;
4   import java.io.*;
5   import java.util.*;
6
7   import javax.imageio.*;
8   import javax.imageio.stream.*;
9   import javax.swing.*;
10  import javax.swing.filechooser.*;
11
12  /**
13   * В этом фрейме отображаются загружаемые изображения.
14   * Для загрузки и сохранения изображений в файл
15   * предоставляются отдельные пункты меню
16   */
17  public class ImageIOFrame extends JFrame
18  {
19    private static final int DEFAULT_WIDTH = 400;
20    private static final int DEFAULT_HEIGHT = 400;
21
22    private static Set<String> writerFormats =
23            getWriterFormats();
24
25    private BufferedImage[] images;
26
```

```
27    public ImageIOFrame()
28    {
29      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
30
31       var fileMenu = new JMenu("File");
32       var openItem = new JMenuItem("Open");
33      openItem.addActionListener(event -> openFile());
34      fileMenu.add(openItem);
35
36      var saveMenu = new JMenu("Save");
37      fileMenu.add(saveMenu);
38      Iterator<String> iter = writerFormats.iterator();
39      while (iter.hasNext())
40      {
41        final String formatName = iter.next();
42        var formatItem = new JMenuItem(formatName);
43        saveMenu.add(formatItem);
44        formatItem.addActionListener(event ->
45                  saveFile(formatName));
46      }
47
48      var exitItem = new JMenuItem("Exit");
49      exitItem.addActionListener(event -> System.exit(0));
50      fileMenu.add(exitItem);
51
52      var menuBar = new JMenuBar();
53      menuBar.add(fileMenu);
54      setJMenuBar(menuBar);
55    }
56
57    /**
58     * Открыть файл и загрузить изображения
59     */
60    public void openFile()
61    {
62      var chooser = new JFileChooser();
63      chooser.setCurrentDirectory(new File("."));
64      String[] extensions =
65               ImageIO.getReaderFileSuffixes();
66      chooser.setFileFilter(new FileNameExtensionFilter(
67                           "Image files", extensions));
68      int r = chooser.showOpenDialog(this);
69      if (r != JFileChooser.APPROVE_OPTION) return;
70      File f = chooser.getSelectedFile();
71      Box box = Box.createVerticalBox();
72      try
73      {
74        String name = f.getName();
75        String suffix = name.substring(
76                       name.lastIndexOf('.') + 1);
77        Iterator<ImageReader> iter =
78                 ImageIO.getImageReadersBySuffix(suffix);
79        ImageReader reader = iter.next();
80        ImageInputStream imageIn =
81                 ImageIO.createImageInputStream(f);
82        reader.setInput(imageIn);
```

```
83        int count = reader.getNumImages(true);
84        images = new BufferedImage[count];
85        for (int i = 0; i < count; i++)
86        {
87          images[i] = reader.read(i);
88          box.add(new JLabel(new ImageIcon(images[i])));
89        }
90      }
91      catch (IOException e)
92      {
93        JOptionPane.showMessageDialog(this, e);
94      }
95      setContentPane(new JScrollPane(box));
96      validate();
97    }
98
99    /**
100    * Сохранить текущее изображение в файле
101    * @param formatName Формат файла
102    */
103   public void saveFile(final String formatName)
104   {
105     if (images == null) return;
106     Iterator<ImageWriter> iter =
107        ImageIO.getImageWritersByFormatName(formatName);
108     ImageWriter writer = iter.next();
109     var chooser = new JFileChooser();
110     chooser.setCurrentDirectory(new File("."));
111     String[] extensions = writer
112           .getOriginatingProvider().getFileSuffixes();
113     chooser.setFileFilter(new FileNameExtensionFilter(
114                           "Image files", extensions));
115
116     int r = chooser.showSaveDialog(this);
117     if (r != JFileChooser.APPROVE_OPTION) return;
118     File f = chooser.getSelectedFile();
119     try
120     {
121       ImageOutputStream imageOut =
122            ImageIO.createImageOutputStream(f);
123        writer.setOutput(imageOut);
124
125        writer.write(new IIOImage(
126                    images[0], null, null));
127        for (int i = 1; i < images.length; i++)
128        {
129          var iioImage =
130              new IIOImage(images[i], null, null);
131          if (writer.canInsertImage(i))
132              writer.writeInsert(i, iioImage, null);
133        }
134     }
135     catch (IOException e)
136     {
137       JOptionPane.showMessageDialog(this, e);
138     }
```

```
139    }
140
141    /**
142     * Получает ряд предпочтительных названий
143     * форматов из всех средств записи.
144     * Предпочтительным считается первое название
145     * формата, указываемое средством записи
146     * @return Возвращает ряд названий форматов
147     */
148    public static Set<String> getWriterFormats()
149    {
150      var writerFormats = new TreeSet<>();
151      var formatNames = new TreeSet<>(
152      Arrays.asList(ImageIO.getWriterFormatNames()));
153      while (formatNames.size() > 0)
154      {
155        String name = formatNames.iterator().next();
156        Iterator<ImageWriter> iter =
157             ImageIO.getImageWritersByFormatName(name);
158        ImageWriter writer = iter.next();
159        String[] names = writer
160              .getOriginatingProvider().getFormatNames();
161        String format = names[0];
162        if (format.equals(format.toLowerCase()))
163        format = format.toUpperCase();
164        writerFormats.add(format);
165        formatNames.removeAll(Arrays.asList(names));
166      }
167      return writerFormats;
168    }
169 }
```

javax.imageio.ImageIO 1.4

- **static BufferedImage read(File *input*)**
- **static BufferedImage read(InputStream *input*)**
- **static BufferedImage read(URL *input*)**

  Читают изображение из указанного источника ввода данных.
- **static boolean write(RenderedImage *image*, String *formatName*, File *output*)**
- **static boolean write(RenderedImage *image*, String *formatName*, OutputStream *output*)**

  Записывают изображение в указанное место назначения вывода данных. Возвращают логическое значение **false**, если не удалось найти соответствующее средство записи.
- **static Iterator<ImageReader> getImageReadersByFormatName(String *formatName*)**
- **static Iterator<ImageReader> getImageReadersBySuffix(String *fileSuffix*)**
- **static Iterator<ImageReader> getImageReadersByMIMEType(String *mimeType*)**

javax.imageio.ImageIO 1.4 *(окончание)*

- **static Iterator<ImageWriter> getImageWritersByFormatName(String *formatName*)**
- **static Iterator<ImageWriter> getImageWritersBySuffix(String *fileSuffix*)**
- **static Iterator<ImageWriter> getImageWritersByMIMEType(String *mimeType*)**

  Получают все средства чтения или записи, поддерживающие указанный формат (например, JPG), расширение файла (например, **jpg**) или тип MIME (например, **mage/jpeg**).
- **static String[] getReaderFormatNames()**
- **static String[] getReaderMIMETypes()**
- **static String[] getWriterFormatNames()**
- **static String[] getWriterMIMETypes()**
- **static String[] getReaderFileSuffixes()** 6
- **static String[] getWriterFileSuffixes()** 6

  Получают названия всех форматов, расширений файлов и типов MIME, которые поддерживаются средствами чтения и записи.
- **ImageInputStream createImageInputStream(Object *input*)**
- **ImageOutputStream createImageOutputStream(Object *input*)**

  Создают поток ввода или вывода изображений, исходя из указанного объекта. В качестве такого объекта может быть указан файл, поток ввода-вывода, экземпляр класса **RandomAccessFile** или другой объект, для которого существует соответствующий поставщик услуг. Если для манипулирования указанным объектом не зарегистрирован требующийся поставщик услуг, возвращается пустое значение **null**.

javax.imageio.ImageReader 1.4

- **void setInput(Object *input*)**
- **void setInput(Object *input*, boolean *seekForwardOnly*)**

  Устанавливают источник ввода данных для средства чтения.

  Параметры: ***input*** Объект типа **ImageInputStream** или другой объект, приемлемый для данного средства чтения

  seekForwardOnly Принимает логическое значение **true** для чтения только в прямом направлении. По умолчанию средство чтения использует произвольный доступ и, если требуется, буферизирует данные изображения
- **BufferedImage read(int *index*)**

  Считывает изображение по указанному индексу, начиная с нуля. Если такое изображение отсутствует, генерируется исключение типа **IndexOutOfBoundsException**.

javax.imageio.ImageReader 1.4 *(окончание)*

- **int getNumImages(boolean *allowSearch*)**

  Получает количество изображений для данного средства чтения. Если параметр ***allowSearch*** принимает логическое значение **false** и количество изображений не может быть определено без предварительного просмотра, возвращается значение **-1**. Если же параметр ***allowSearch*** принимает логическое значение **true** и средство чтения находится в режиме просмотра данных только в прямом направлении, генерируется исключение типа **IllegalStateException**.
- **int getNumThumbnails(int *index*)**

  Получает количество миниатюрных видов изображения по указанному индексу.
- **BufferedImage readThumbnail(int *index*, int *thumbnailIndex*)**

  Получает по индексу ***thumbnailIndex*** миниатюрный вид изображения, указанного по индексу ***index***.
- **int getWidth(int *index*)**
- **int getHeight(int *index*)**

  Получают ширину и высоту изображения. Если изображение не найдено, генерируется исключение типа **IndexOutOfBoundsException**.
- **ImageReaderSpi getOriginatingProvider()**

  Получает поставщика услуг, создавшего данное средство чтения.

javax.imageio.spi.IIOServiceProvider 1.4

- **String getVendorName()**
- **String getVersion()**

  Получают имя и версию данного поставщика услуг.

javax.imageio.spi.ImageReaderWriterSpi 1.4

- **String[] getFormatNames()**
- **String[] getFileSuffixes()**
- **String[] getMIMETypes()**

  Получают названия форматов, расширения файлов и типа MIME, которые поддерживаются средствами чтения или записи, созданными данным поставщиком услуг.

javax.imageio.ImageWriter 1.4

- **void setOutput(Object *output*)**

  Устанавливает место назначения вывода для данного средства записи.

javax.imageio.ImageWriter 1.4 *(окончание)*

Параметры: ***output*** Объект типа **ImageOutputStream** или другой объект, приемлемый для данного средства записи

- **void write(IIOImage *image*)**
- **void write(RenderedImage *image*)**

  Записывают одиночное изображение по месту вывода.
- **void writeInsert(int *index*, IIOImage *image*, ImageWriteParam *param*)**

  Записывает изображение в файл с несколькими изображениями.
- **boolean canInsertImage(int *index*)**

  Возвращает логическое значение **true**, если в файл можно ввести изображение по указанному индексу.
- **ImageWriterSpi getOriginatingProvider()**

  Получает поставщика услуг, создавшего данное средство записи.

javax.imageio.IIOImage 1.4

- **IIOImage(RenderedImage *image*, List *thumbnails*, IIOMetadata *metadata*)**

  Создает объект типа **IIOImage**, исходя из указанного изображения, необязательных миниатюрных видов и метаданных.

11.4.2. Манипулирование изображениями

Допустим, требуется улучшить внешний вид какого-нибудь изображения. Для этого следует получить доступ к отдельным пикселям изображения и заменить их другими. Не исключено также, что придется сформировать новые пиксели заново, например, для отображения результатов физических измерений или математических расчетов. Класс `BufferedImage` дает возможность манипулировать пикселями изображения, а классы, реализующие интерфейс `BufferedImageOp`, — преобразовывать изображения.

НА ЗАМЕТКУ! В версии JDK 1.0 предоставлялась совершенно другая и намного более сложная среда для обработки изображений, которая была оптимизирована для *пошагового воспроизведения* изображений, построчно загружавшихся из Интернета. Но манипулировать изображениями в такой среде было совсем не просто. Поэтому в данной книге эта среда не рассматривается.

11.4.2.1. Формирование растровых изображений

Большинство обрабатываемых изображений считываются из файла и формируются аппаратными средствами (например, цифровой камерой или сканером) или же программными (например, графическими приложениями). В этом разделе рассматривается совершенно другая методика формирования растровых изображений, предполагающая построение в каждый отдельный момент времени только одного пикселя.

Чтобы сформировать растровое изображение, следует построить сначала объект типа BufferedImage обычным способом:

```
image = new BufferedImage(width, height,
                          BufferedImage.TYPE_INT_ARGB);
```

Затем необходимо вызвать метод getRaster(), как показано ниже, чтобы получить объект типа WritableRaster, который служит для доступа к пикселям растрового изображения с целью внести в них изменения.

```
WritableRaster raster = image.getRaster();
```

Метод setPixel() позволяет задавать значение цвета отдельного пикселя. Но когда он применяется, возникает следующее осложнение: пикселю нельзя непосредственно присвоить значение цвета типа Color. Необходимо знать, каким образом цвета определяются в буферизированном изображении, что зависит от его *типа*. Так, если изображение относится к типу TYPE_INT_ARGB, то каждый пиксель описывается четырьмя значениями (в пределах от 0 до 255): для красной, зеленой и синей составляющих, а также для альфа-канала. Эти значения необходимо предоставить в виде массива из четырех элементов целого типа, как показано ниже. В прикладном интерфейсе Java 2D API такие значения называются *выборочными значениями* цвета пикселя.

```
int[] black = { 0, 0, 0, 255 };
raster.setPixel(i, j, black);
```

ВНИМАНИЕ! Метод **setPixel()** может принимать в качестве параметра массив типа **float[]** или **double[]**. Следует, однако, иметь в виду, что значения цвета, которыми заполняется этот массив, *не* должны быть нормализованными в пределах от **0** до **1.0**, как показано ниже.

```
float[] red = { 1.0F, 0.0F, 0.0F, 1.0F };
raster.setPixel(i, j, red); // ОШИБКА!
```

Независимо от своего типа массив все равно должен быть заполнен значениями цвета в пределах от **0** до **255**.

Для установки значений группы прямоугольных пикселей в качестве параметров метода setPixels() можно указать исходное положение, ширину и высоту прямоугольника, а также массив с выборочными значениями цвета этой группы пикселей. Так, если изображение относится к типу TYPE_INT_ARGB, сначала следует указать величины красной, зеленой и синей составляющих и значение прозрачности в альфа-канале для первого пикселя, а затем те же самые данные для второго пикселя и т.д. В приведенном ниже фрагменте кода показано, как это делается.

```
var pixels = new int[4 * width * height];
// значение красной составляющей цвета 1-го пикселя:
pixels[0] = . . .
// значение зеленой составляющей цвета 1-го пикселя:
pixels[1] = . . .
// значение синей составляющей цвета 1-го пикселя:
pixels[2] = . . .
// значение прозрачности в альфа-канале 1-го пикселя:
```

```
pixels[3] = . . .
. . .
raster.setPixels(x, y, width, height, pixels);
```

Для чтения значения цвета пикселя служит приведенный ниже метод `getPixel()`, возвращающий массив из четырех целых чисел.

```
var sample = new int[4];
raster.getPixel(x, y, sample);
var c = new Color(sample[0], sample[1],
                  sample[2], sample[3]);
```

С помощью метода `getPixels()` можно прочитать значения цвета группы пикселей следующим образом:

```
raster.getPixels(x, y, width, height, samples);
```

Методы `getPixel()` и `setPixel()` можно использовать и для другого типа изображения, если известен способ представления в нем значений цвета пикселей. Иными словами, чтобы применять эти методы, необходимо знать способ кодирования выборочных значений цвета пикселей в конкретном типе изображения.

Манипулировать изображением становится сложнее, если его тип заранее неизвестен. У каждого типа растровых изображений имеется своя *цветовая модель*, в соответствии с которой выборочные значения цвета пикселей можно привести к стандартной цветовой модели RGB.

НА ЗАМЕТКУ! На самом деле цветовая модель RGB не является стандартом. Точное представление цветов зависит от характеристик используемого устройства отображения. Цифровые камеры, сканеры, мониторы и жидкокристаллические дисплеи имеют совершенно разные характеристики. В результате одно и то же значение цвета RGB выглядит по-разному на разных устройствах отображения. Международный консорциум по цвету (International Color Consortium — ICC; **`http://www.color.org`**) рекомендует сопровождать все цветовые данные цветовым *профилем IIC*, который определяет степень соответствия цветов стандартной форме их представления по спецификации 1931 CIE XYZ. Эта спецификация разработана Международной комиссией по освещению (Commission Internationale de l'Eclairage — CIE; **`http://www.cie.co.at`**), обеспечивающей техническую поддержку всех видов освещения и цвета. Она регламентирует стандартный метод представления всех цветов, которые глаз человека может воспринимать на основе трех гипотетических координат X, Y, Z. (Подробнее эта спецификация описана в главе 28 упоминавшейся ранее книги *Computer Graphics: Principles and Practice, Third Edititon* Джеймса Фоли, Андриса ван Дама, Стивена Фэйнера и др.)

Но цветовые профили ICC имеют очень сложную структуру. Поэтому наряду с ними используется более простой стандарт sRGB (**`https://www.w3.org/Images/Color/sRGB.html`**), который устанавливает точное соответствие значений цветов RGB спецификации 1931 CIE XYZ. Он разработан специально для стандартных цветных мониторов и используется в прикладном интерфейсе Java 2D API для преобразования цветов RGB в значения цветов других цветовых моделей и пространств.

Для получения цветовой модели вызывается метод `getColorModel()`:

```
ColorModel model = image.getColorModel();
```

Для определения цвета пикселя следует вызвать приведенный ниже метод `getDataElements()` из класса `Raster`, который возвращает объект типа `Object` с описанием цвета в соответствии с конкретной цветовой моделью.

```
Object data = raster.getDataElements(x, y, null);
```

НА ЗАМЕТКУ! Объект, который возвращается методом **`getDataElements()`**, в действительности является массивом выборочных значений цвета пикселя. Это объясняет, почему метод **`getDataElements()`** называется именно так, а не иначе. Хотя для обработки данного объекта знать об этом совсем не обязательно.

С помощью цветовой модели можно преобразовать полученный объект в стандартные значения цвета ARGB. Метод `getRGB()` возвращает значение типа `int`, состоящее из значений степени прозрачности в альфа-канале, красной, зеленой и синей составляющих цвета, упакованных в четыре блока по 8 бит в каждом. Из этого целочисленного составного значения цвета с помощью конструктора `Color(int argb, boolean hasAlpha)` можно создать объект типа `Color`:

```
int argb = model.getRGB(data);
var color = new Color(argb, true);
```

Если же требуется установить определенный цвет пикселя, то описанные выше действия следует выполнить в обратном порядке. Метод `getRGB()` из класса `Color` возвращает целочисленное значение, состоящее из значений степени прозрачности в альфа-канале, красной, зеленой и синей составляющих цвета. Именно его следует указать в качестве параметра при вызове метода `getDataElements()` из класса `ColorModel`, который, в свою очередь, возвратит объект с описанием цвета пикселя в используемой цветовой модели. Далее этот объект следует передать методу `setDataElements()` из класса `WritableRaster`, как показано ниже.

```
int argb = color.getRGB();
Object data = model.getDataElements(argb, null);
raster.setDataElements(x, y, data);
```

Чтобы проиллюстрировать, каким образом растровое изображение формируется из отдельных пикселей, рассмотрим пример рисования фрактального множества Мандельброта, отдавая дань традиции. Множество Мандельброта образуется путем связывания каждой точки плоскости с некоторой последовательностью цифр. Если последовательность ограничена, точка отмечается цветом. Неокрашенные точки плоскости связаны с неограниченными последовательностями цифр и остаются прозрачными (рис. 11.56).

Ниже показано, как построить простейшее множество Мандельброта. Сначала необходимо отыскать для каждой точки с координатами (a, b) последовательность, которая начинается с выражения (x, y) = (0, 0), и применить к ней следующие формулы:

$$x_{new} = x^2 - y^2 + a$$
$$y_{new} = 2 \cdot x \cdot y + b$$

Если значение x или y больше 2, то последовательность будет превращаться в бесконечную и окрашиваться цветом будут только те пиксели, которые соответствуют точкам с координатами (a, b), ведущим к ограниченной последовательности. (Формулы для вычисления числовых последовательностей взяты из области математики комплексных чисел в их исходном виде.)

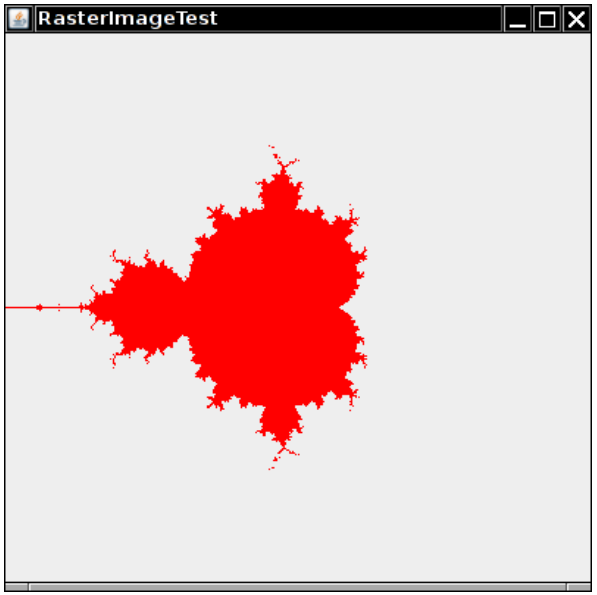


Рис. 11.56. Множество Мандельброта

В примере программы из листинга 11.21 демонстрируется применение класса `ColorModel` для преобразования значений цвета типа `Color` в выборочные данные цвета пикселей. Этот процесс не зависит от типа изображения. Ради интереса попробуйте поэкспериментировать, изменив тип цвета для буферизированного изображения на `TYPE_BYTE_GRAY`. В исходный код данной программы не нужно вносить больше никаких других изменений, поскольку цветовая модель изображения автоматически преобразует цвета пикселей в выборочные значения.

Листинг 11.21. Исходный код из файла **`rasterImage/RasterImageFrame.java`**

```
1  package rasterImage;
2
3  import java.awt.*;
4  import java.awt.image.*;
5  import javax.swing.*;
6
7  /**
8   * В этом фрейме показывается изображение
9   * множества Мандельброта
10  */
11 public class RasterImageFrame extends JFrame
12 {
13   private static final double XMIN = -2;
14   private static final double XMAX = 2;
15   private static final double YMIN = -2;
16   private static final double YMAX = 2;
17   private static final int MAX_ITERATIONS = 16;
18   private static final int IMAGE_WIDTH = 400;
19   private static final int IMAGE_HEIGHT = 400;
```

```
20
21   public RasterImageFrame()
22   {
23     BufferedImage image = makeMandelbrot(
24                           IMAGE_WIDTH, IMAGE_HEIGHT);
25     add(new JLabel(new ImageIcon(image)));
26     pack();
27   }
28
29   /**
30    * Формирует изображение множества Мандельброта
31    * @param width Ширина изображения
32    * @parah height Высота изображения
33    * @return Изображение
34   */
35   public BufferedImage makeMandelbrot(
36                         int width, int height)
37   {
38     var image = new BufferedImage(width, height,
39                     BufferedImage.TYPE_INT_ARGB);
40     WritableRaster raster = image.getRaster();
41     ColorModel model = image.getColorModel();
42
43     Color fractalColor = Color.red;
44     int argb = fractalColor.getRGB();
45     Object colorData = model.getDataElements(argb, null);
46
47     for (int i = 0; i < width; i++)
48       for (int j = 0; j < height; j++)
49       {
50         double a = XMIN + i * (XMAX - XMIN) / width;
51         double b = YMIN + j * (YMAX - YMIN) / height;
52         if (!escapesToInfinity(a, b))
53             raster.setDataElements(i, j, colorData);
54       }
55     return image;
56   }
57
58   private boolean escapesToInfinity(double a, double b)
59   {
60     double x = 0.0;
61     double y = 0.0;
62     int iterations = 0;
63     while (x <= 2 && y <= 2
64                   && iterations < MAX_ITERATIONS)
65     {
66       double xnew = x * x - y * y + a;
67       double ynew = 2 * x * y + b;
68       x = xnew;
69       y = ynew;
70       iterations++;
71     }
72     return x > 2 || y > 2;
73   }
74 }
```

java.awt.image.BufferedImage 1.2

- **BufferedImage(int *width*, int *height*, int *imageType*)**
  Создает объект буферизированного изображения.
  Параметры: ***width***, ***height*** Размеры изображения
  imageType Тип изображения. К числу наиболее распространенных относятся следующие типы изображений: **TYPE_INT_RGB**, **TYPE_INT_ARGB**, **TYPE_BYTE_GRAY** и **TYPE_BYTE_INDEXED**
- **ColorModel getColorModel()**
  Возвращает цветовую модель данного буферизированного изображения.
- **WritableRaster getRaster()**
  Возвращает растр для доступа к пикселям данного буферизированного изображения с целью их изменения.

java.awt.image.Raster 1.2

- **Object getDataElements(int *x*, int *y*, Object *data*)**
  Возвращает для точки растра выборочные данные в виде массива, тип элементов и длина которого зависят от цветовой модели. Если параметр ***data*** не принимает пустое значение **null**, то предполагается, что это значение является массивом для хранения выборочных данных, причем массив заполнен. Если параметр ***data*** принимает пустое значение **null**, то создается новый массив. Тип его элементов и длина зависят от цветовой модели.
- **int[] getPixel(int *x*, int *y*, int[] *sampleValues*)**
- **float[] getPixel(int *x*, int *y*, float[] *sampleValues*)**
- **double[] getPixel(int *x*, int *y*, double[] *sampleValues*)**
- **int[] getPixels(int *x*, int *y*, int *width*, int *height*, int[] *sampleValues*)**
- **float[] getPixels(int *x*, int *y*, int *width*, int *height*, float[] *sampleValues*)**
- **double[] getPixels(int *x*, int *y*, int *width*, int *height*, double[] *sampleValues*)**
  Возвращают выборочные значения для точки растра или группы точек, составляющих прямоугольную область. Эти значения помещаются в массив, длина которого зависит от конкретной цветовой модели. Если параметр ***sampleValues*** принимает пустое значение **null**, то создается новый массив. Применяются только в том случае, если заранее известно, каким образом выборочные значения определяются для цветовой модели.

java.awt.image.WritableRaster 1.2

- **void setDataElements(int *x*, int *y*, Object *data*)**
  Задает выборочные данные для точки растра. Параметр ***data*** обозначает массив выборочных значений цвета пикселя. Тип элементов и длина этого массива зависят от конкретной цветовой модели.

java.awt.image.WritableRaster 1.2 *(окончание)*

- **void setPixel(int *x*, int *y*, int[] *sampleValues*)**
- **void setPixel(int *x*, int *y*, float[] *sampleValues*)**
- **void setPixel(int *x*, int *y*, double[] *sampleValues*)**
- **void setPixels(int *x*, int *y*, int *width*, int *height*, int[] *sampleValues*)**
- **void setPixels(int *x*, int *y*, int *width*, int *height*, float[] *sampleValues*)**
- **void setPixels(int *x*, int *y*, int *width*, int *height*, double[] *sampleValues*)**

  Устанавливают выборочные значения для точки растра или группы точек, составляющих прямоугольную область. Применяются только в том случае, если заранее известен порядок кодировки выборочных значений для цветовой модели.

java.awt.image.ColorModel 1.2

- **int getRGB(Object *data*)**

  Возвращает значение цвета ARGB, которое соответствует выборочным данным, переданным в массиве ***data***, тип элементов и длина которого зависят от конкретной цветовой модели.
- **Object getDataElements(int *argb*, Object *data*)**

  Возвращает выборочные данные для указанного значения цвета. Если параметр ***data*** не принимает пустое значение **null**, то предполагается, что это массив, имеющий подходящую длину для хранения данных и заполненный ими. Если параметр ***data*** принимает пустое значение **null**, то создается новый массив. Этот массив заполняется выборочными данными цвета пикселя. Тип его элементов и длина массива зависят от цветовой модели.

java.awt.Color 1.0

- **Color(int *argb*, boolean *hasAlpha*)** 1.2

  Формирует цвет по указанному составному значению ARGB, если параметр ***hasAlpha*** принимает логическое значение **true**, или же по указанному стандартному значению RGB, если параметр ***hasAlpha*** принимает логическое значение **false**.
- **int getRGB()**

  Возвращает значение ARGB, соответствующее данному цвету.

11.4.2.2. Фильтрация изображений

В предыдущем разделе рассматривался процесс формирования растрового изображения заново. Но нередко требуется обработать уже имеющееся изображение. Безусловно, данные изображения можно сначала прочитать, используя методы `getPixel()`/`getDataElements()`, представленные в предыдущем разделе, а затем преобразовать их и записать обратно. Но, к счастью, в прикладном интерфейсе Java 2D API поддерживается целый ряд *фильтров*, автоматически выполняющих многие рутинные операции обработки изображений.

Все классы, выполняющие операции манипулирования изображениями, реализуют интерфейс `BufferedImageOp`. Для преобразования одного изображения в другое после создания объекта требующейся операции достаточно вызвать метод `filter()` следующим образом:

```
BufferedImageOp op = . . .;
BufferedImage filteredImage = new BufferedImage(
                    image.getHeight(), image.getType());
op.filter(image, filteredImage);
```

Некоторые методы, например `op.filter(image, image)`, могут непосредственно выполнять преобразование изображений, но большинство остальных методов не способны на это. Ниже перечислены пять классов, реализующих интерфейс `BufferedImageOp`.

```
AffineTransformOp
RescaleOp
LookupOp
ColorConvertOp
ConvolveOp
```

В частности, класс `AffineTransformOp` выполняет аффинное преобразование пикселей. Например, в приведенном ниже фрагменте кода показано, как повернуть изображение вокруг его центра.

```
AffineTransform transform = AffineTransform
            .getRotateInstance(Math.toRadians(angle),
                               image.getWidth() / 2,
                               image.getHeight() / 2);
var op = new AffineTransformOp(transform, interpolation);
op.filter(image, filteredImage);
```

При вызове конструктора класса `AffineTransformOp` в качестве параметров следует указать аффинное преобразование и алгоритм *интерполяции*. Интерполяция требуется для определения местоположения пикселей в целевом изображении, если исходные пиксели оказываются после преобразования где-то между целевыми пикселями.

Например, после вращения исходные пиксели точно совпадают с целевыми пикселями. Для интерполяции используются три алгоритма: бикубический (`AffineTransformOp.TYPE_BICUBIC`), билинейный (`AffineTransformOp.TYPE_BILINEAR`) и ступенчатый, или ближайшего соседа (`AffineTransformOp.TYPE_NEAREST_NEIGHBOR`). Билинейная интерполяция выполняется дольше, но позволяет добиться более качественного внешнего вида преобразованного изображения, чем два других алгоритма интерполяции.

В примере программы из листинга 11.22 изображение поворачивается на 5° (рис. 11.57). В частности, класс `RescaleOp` выполняет для каждой составляющей цвета изображения следующую операцию по изменению масштаба:

$$x_{new} = a \cdot x + b$$

(На составляющие прозрачности из альфа-канала эта операция не распространяется.) В результате изменения масштаба на величину `a > 1` увеличивается яркость изображения. Объект типа `RescaleOp` создается по указанным параметрам масштабирования и дополнительным, но не обязательным параметрам

рисования. В примере программы из листинга 11.22 для этой цели используется приведенный ниже фрагмент кода. Кроме того, отдельные величины изменения масштаба можно предоставить для каждой составляющей цвета, как поясняется далее при описании прикладного интерфейса API.

```
float a = 1.1f;
float 20.0f;
var op = new RescaleOp(a, b, null);
```

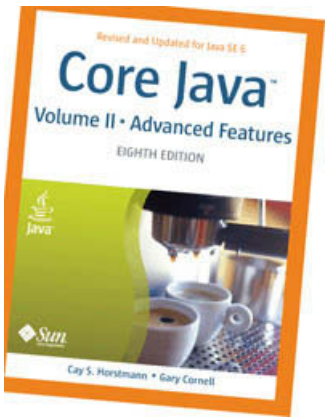


Рис. 11.57. Вращение изображения

Конструктор класса `LookupOp` позволяет указать произвольное отображение выборочных значений цвета. Для этого достаточно предоставить таблицу, в которой указаны правила отображения каждого выборочного значения цвета. В рассматриваемом здесь примере формируется *негатив* исходного изображения. Для этого каждая составляющая цвета `c` заменяется на `255 - c`.

В качестве параметров конструктору класса `LookupOp` требуется указать объект типа `LookupTable` и таблицу с необязательными указаниями по воспроизведению. Класс `LookupTable` является абстрактным и имеет два конкретных подкласса: `ByteLookupTable` и `ShortLookupTable`. На первый взгляд, для целей преобразования было бы достаточно и класса `ByteLookupTable`, поскольку значения цвета RGB представлены в виде байтов. Но из-за программной ошибки, подробно описанной в документации, доступной по адресу https://bugs.java.com/bugdatabase/view_bug.do?bug_id=6183251, вместо него в данном примере используется класс `ShortLookupTable`. В приведенном ниже фрагменте кода показано, каким образом формируется операция табличного поиска для преобразования цветного изображения в негатив.

```
var negative = new short[256];
for (int i = 0; i < 256; i++)
  negative[i] = (short) (255 - i);
var table = new ShortLookupTable(0, negative);
var op = new LookupOp(table, null);
```

Операция табличного поиска выполняется над каждой составляющей цвета в отдельности, но не над составляющими прозрачности в альфа-канале. Для преобразования каждой составляющей цвета можно предоставить и другие таблицы поиска (подробнее об этом см. далее в описании прикладного интерфейса API).

НА ЗАМЕТКУ! Класс **LookupOp** нельзя применять к изображениям с индексированной цифровой моделью, потому что в таких изображениях каждое выборочное значение цвета пикселей задается в виде смещения в палитре цветов.

Для преобразования цветового пространства служит класс `ColorConvertOp`, который здесь не рассматривается. Для наиболее глубоких преобразований предназначен класс `ConvolveOp`, который выполняет математическую операцию *свертки*. Мы не будем углубляться в математические подробности выполнения операции свертки, но понять основной ее принцип действия несложно. В качестве примера рассмотрим фильтр размытия, действие которого показано на рис. 11.58.

Рис. 11.58. Размытие изображения

Размытие достигается путем замены каждого пикселя *средним* значением этого пикселя и соседних с ним восьми пикселей. Интуитивно понятно, почему эта операция делает изображение размытым, а с математической точки зрения получение среднего значения может быть представлено как операция свертки со следующим ядром:

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

Ядро свертки представляет собой матрицу с весами соседних значений. Например, приведенное выше ядро приводит к размытию изображения. А представленное ниже ядро выполняет операцию *определения краев*, т.е. определяет участки, на которых происходит изменение цвета. Алгоритм определения краев играет важную роль в обработке фотоизображений (рис. 11.59).

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Рис. 11.59. Определение краев и инверсия

Чтобы сформировать операцию свертки, необходимо сначала составить массив значений для ядра свертки и создать объект типа `Kernel`. Затем на основе этого ядра следует создать объект типа `ConvolveOp`, чтобы использовать его для фильтрации изображения следующим образом:

```
float[] elements =
  {
    0.0f, -1.0f, 0.0f,
    -1.0f, 4.f, -1.0f,
    0.0f, -1.0f, 0.0f
  };
var kernel = new Kernel(3, 3, elements);
var op = new ConvolveOp(kernel);
op.filter(image, filteredImage);
```

В примере программы из листинга 11.22 пользователю предоставляется возможность загружать файлы формата GIF и JPEG и выполнять над ними описанные выше операции. Простота кода этой программы объясняется тем, что в прикладном интерфейсе Java 2D API предусмотрены весьма эффективные средства для обработки растровых изображений.

Листинг 11.22. Исходный код из файла **`ImageProcessing/ImageProcessingFrame.java`**

```
1   package imageProcessing;
2
3   import java.awt.*;
4   import java.awt.geom.*;
5   import java.awt.image.*;
6   import java.io.*;
7
8   import javax.imageio.*;
9   import javax.swing.*;
10  import javax.swing.filechooser.*;
11
```

```
12  /**
13   * Этот фрейм содержит меню для загрузки изображения
14   * и выбора различных его преобразований, а также
15   * компонент для показа итогового изображения
16   */
17  public class ImageProcessingFrame extends JFrame
18  {
19    private static final int DEFAULT_WIDTH = 400;
20    private static final int DEFAULT_HEIGHT = 400;
21
22    private BufferedImage image;
23
24    public ImageProcessingFrame()
25    {
26      setTitle("ImageProcessingTest");
27      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
28
29      add(new JComponent()
30        {
31          public void paintComponent(Graphics g)
32          {
33            if (image != null) g.drawImage(image, 0, 0, null);
34          }
35        });
36
37      var fileMenu = new JMenu("File");
38      var openItem = new JMenuItem("Open");
39      openItem.addActionListener(event -> openFile());
40      fileMenu.add(openItem);
41
42      var exitItem = new JMenuItem("Exit");
43      exitItem.addActionListener(event -> System.exit(0));
44      fileMenu.add(exitItem);
45
46      var editMenu = new JMenu("Edit");
47      var blurItem = new JMenuItem("Blur");
48      blurItem.addActionListener(event ->
49        {
50          float weight = 1.0f / 9.0f;
51          float[] elements = new float[9];
52          for (int i = 0; i < 9; i++)
53            elements[i] = weight;
54          convolve(elements);
55        });
56      editMenu.add(blurItem);
57
58      var sharpenItem = new JMenuItem("Sharpen");
59      sharpenItem.addActionListener(event ->
60        {
61          float[] elements =
62                  { 0.0f, -1.0f, 0.0f, -1.0f, 5.f,
63                    -1.0f, 0.0f, -1.0f, 0.0f };
64          convolve(elements);
65        });
```

```
66      editMenu.add(sharpenItem);
67
68      var brightenItem = new JMenuItem("Brighten");
69      brightenItem.addActionListener(event ->
70        {
71          float a = 1.1f;
72          float b = 20.0f;
73          var op = new RescaleOp(a, b, null);
74          filter(op);
75        });
76      editMenu.add(brightenItem);
77
78      var edgeDetectItem = new JMenuItem("Edge detect");
79      edgeDetectItem.addActionListener(event ->
80        {
81          float[] elements =
82                  { 0.0f, -1.0f, 0.0f, -1.0f, 4.f,
83                   -1.0f, 0.0f, -1.0f, 0.0f };
84          convolve(elements);
85        });
86      editMenu.add(edgeDetectItem);
87
88      var negativeItem = new JMenuItem("Negative");
89      negativeItem.addActionListener(event ->
90        {
91          short[] negative = new short[256 * 1];
92          for (int i = 0; i < 256; i++)
93            negative[i] = (short) (255 - i);
94          var table = new ShortLookupTable(0, negative);
95          var op = new LookupOp(table, null);
96          filter(op);
97        });
98      editMenu.add(negativeItem);
99
100     var rotateItem = new JMenuItem("Rotate");
101     rotateItem.addActionListener(event ->
102       {
103         if (image == null) return;
104         var transform = AffineTransform
105             .getRotateInstance(Math.toRadians(5),
106                                image.getWidth() / 2,
107                                image.getHeight() / 2);
108         var op = new AffineTransformOp(transform,
109                        AffineTransformOp.TYPE_BICUBIC);
110         filter(op);
111       });
112     editMenu.add(rotateItem);
113
114     var menuBar = new JMenuBar();
115     menuBar.add(fileMenu);
116     menuBar.add(editMenu);
117     setJMenuBar(menuBar);
118   }
119
```

```
120   /**
121    * Открыть файл и загрузить изображение
122    */
123   public void openFile()
124   {
125     var chooser = new JFileChooser(".");
126     chooser.setCurrentDirectory(
127         new File(getClass().getPackage().getName()));
128     Strig[] extensions =
129           ImageIO.getReaderFileSuffixes();
130     chooser.setFileFilter(new FileNameExtensionFilter(
131                           "Image files", extensions));
132     int r = chooser.showOpenDialog(this);
133     if (r != JFileChooser.APPROVE_OPTION) return;
134
135     try
136     {
137       Image img = ImageIO.read(
138                           chooser.getSelectedFile());
139       image = new BufferedImage(img.getWidth(null),
140                                 img.getHeight(null),
141                           BufferedImage.TYPE_INT_RGB);
142       image.getGraphics().drawImage(img, 0, 0, null);
143     }
144     catch (IOException e)
145     {
146       JOptionPane.showMessageDialog(this, e);
147     }
148     repaint();
149   }
150
151   /**
152    * Применить фильтр и перерисовать
153    * @param op Выполняемая операция преобразования
154    */
155   private void filter(BufferedImageOp op)
156   {
157     if (image == null) return;
158     image = op.filter(image, null);
159     repaint();
160   }
161
162   /**
163    * Выполнить свертку и перерисовать
164    * @param elements Ядро свертки(массив из 9
165                       элементов матрицы)
166    */
167   private void convolve(float[] elements)
168   {
169     var kernel = new Kernel(3, 3, elements);
170     ConvolveOp op = new ConvolveOp(kernel);
171     filter(op);
172   }
173 }
```

java.awt.image.BufferedImageOp 1.2

- **BufferedImage filter(BufferedImage *source*, BufferedImage *dest*)**

  Выполняет операцию над исходным изображением ***source*** и сохраняет (а также возвращает) результат в виде изображения ***dest***. Если в качестве параметра ***dest*** указано пустое значение **null**, то создается новое целевое изображение, которое затем возвращается.

java.awt.image.AffineTransformOp 1.2

- **AffineTransformOp(AffineTransform *t*, int *interpolationType*)**

  Создает объект для операции аффинного преобразования. В качестве алгоритма интерполяции может быть указано значение одной из следующих констант: **TYPE_BILINEAR**, **TYPE_BICUBIC** или **TYPE_NEAREST_NEIGHBOR**.

java.awt.image.RescaleOp 1.2

- **RescaleOp(float *a*, float *b*, RenderingHints *hints*)**
- **RescaleOp(float[] *as*, float[] *bs*, RenderingHints *hints*)**

  Создают объект для следующей операции изменения масштаба: $x^{new} = a \cdot x + b$. При использовании первого конструктора все составляющие цвета (кроме составляющей прозрачности из альфа-канала) масштабируются с одинаковыми коэффициентами. При использовании второго конструктора значения предоставляются для каждой составляющей цвета в отдельности, но не затрагивая составляющую прозрачности из альфа-канала, или же для составляющих как цвета, так и прозрачности из альфа-канала.

java.awt.image.LookupOp 1.2

- **LookupOp(LookupTable *table*, RenderingHints *hints*)**

  Создает объект операции поиска для указанной таблицы поиска.

java.awt.image.ByteLookupTable 1.2

- **ByteLookupTable(int *offset*, byte[] *data*)**
- **ByteLookupTable(int *offset*, byte[][] *data*)**

  Создают таблицу поиска для преобразования значений типа **byte**. Значение смещения вычитается из входных данных до преобразования. Значения в первом конструкторе применяются ко всем составляющим цвета, но не к составляющей прозрачности из альфа-канала. При использовании второго конструктора значения предоставляются для каждой составляющей цвета в отдельности, но не затрагивая составляющую прозрачности из альфа-канала, или же для составляющих как цвета, так и прозрачности из альфа-канала.

java.awt.image.ShortLookupTable 1.2

- **ShortLookupTable(int *offset*, short[] *data*)**
- **ShortLookupTable(int *offset*, short[][] *data*)**

  Создают таблицу поиска для преобразования значений типа **short**. Значение смещения вычитается из входных данных до преобразования. Значения в первом конструкторе применяются ко всем составляющим цвета, но не к составляющей прозрачности из альфа-канала. При использовании второго конструктора значения предоставляются для каждой составляющей цвета в отдельности, но не затрагивая составляющую прозрачности из альфа-канала, или же для составляющих как цвета, так и прозрачности из альфа-канала.

java.awt.image.ConvolveOp 1.2

- **ConvolveOp(Kernel *kernel*)**
- **ConvolveOp(Kernel *kernel*, int *edgeCondition*, RenderingHints *hints*)**

  Создают объект для операции свертки. В качестве параметра ***edgeCondition*** может быть указано значение одной из следующих констант: **EDGE_NO_OP** или **EDGE_ZERO_FILL**. С этими значениями следует обращаться особенно внимательно и аккуратно, поскольку у них нет достаточного количества соседних значений для вычисления свертки. По умолчанию устанавливается значение константы **EDGE_ZERO_FILL**.

java.awt.image.Kernel 1.2

- **Kernel(int *width*, int *height*, float[] *matrixElements*)**

  Создает ядро свертки для указанной матрицы.

11.5. Вывод изображений на печать

В последующих разделах поясняется, насколько просто вывести растровое изображение или графический рисунок на печатный лист бумаги. В них также показывается, как организовать вывод растровых изображений и графических рисунков на несколько печатных страниц и сохранить отпечаток в файле формата PostScript.

11.5.1. Вывод графики на печать

В данном разделе рассматривается одна из наиболее распространенных задач — печать двухмерной графики. Разумеется, графическое изображение может содержать фрагменты текста, отформатированного различными шрифтами, или даже полностью состоять из текста. Для вывода двухмерной графики на печать необходимо выполнить следующие действия.

- Предоставить объект, класс которого реализует интерфейс `Printable`.
- Запустить задание на печать.

В интерфейсе `Printable` объявлен единственный метод `print()`:

```
int print(Graphics g, PageFormat format, int page)
```

Этот метод вызывается всякий раз, когда механизм печати форматирует страницу. В прикладном коде воспроизводятся фрагменты текста и изображения, которые должны быть напечатаны в заданном графическом контексте `g`. Формат страницы (параметр `format`) определяет формат бумаги и поля для печати, а номер страницы (параметр `page`) служит для выбора воспроизводимой страницы.

Для запуска задания на печать служит класс `PrinterJob`. Сначала для получения объекта задания на печать вызывается статический метод `getPrinterJob()`, а затем с помощью метода `setPrintable()` указывается печатаемый объект типа `Printable`:

```
Printable canvas = . . .;
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(canvas);
```

ВНИМАНИЕ! Не путайте класс **`PrinterJob`** с устаревшим классом **`PrintJob`**, который управлял процессом печати в версии JDK 1.1.

Прежде чем запускать задание на печать, следует вызвать метод `printDialog()`, чтобы открыть диалоговое окно, приведенное на рис. 11.60. В этом окне пользователю предоставляется возможность выбрать устройство для вывода на печать (если доступно несколько печатающих устройств), указать диапазон печатаемых страниц и сделать прочие настройки печати.

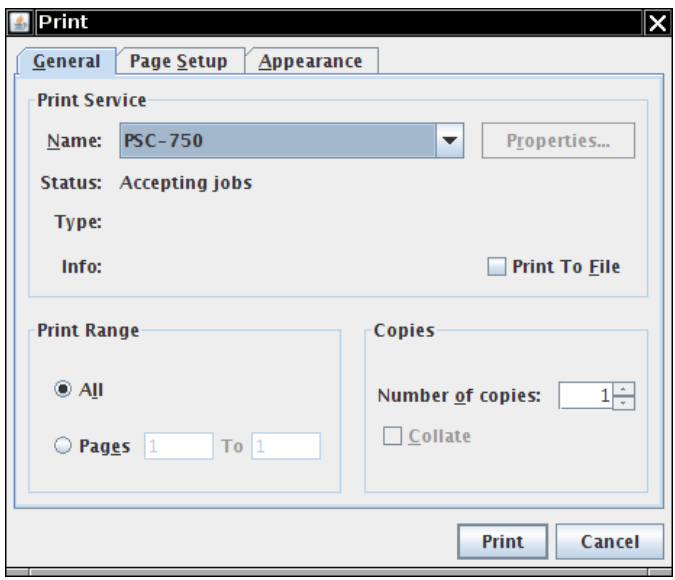


Рис. 11.60. Вид диалогового окна для настройки печати независимо от используемой платформы

Все параметры печатающего устройства обычно собираются в экземпляре класса, реализующего интерфейс `PrintRequestAttributeSet`, например, класса `HashPrintRequestAttributeSet`:

```
var attributes = new HashPrintRequestAttributeSet();
```

Атрибуты печати должны быть переданы методу `printDialog()` в виде объекта `attributes`. Метод `printDialog()` возвращает логическое значение `true`, если пользователь подтверждает выбор параметров печати, или логическое значение `false`, если пользователь отменяет внесенные изменения. В первом случае вызывается метод `print()` из класса `PrinterJob` для запуска задания на печать. Он может генерировать исключение типа `PrinterException`. В приведенном ниже фрагменте кода в общих чертах демонстрируется организация вывода на печать.

```
if (job.printDialog(attributes))
{
  try
  {
    job.print(attributes);
  }
  catch (PrinterException exception)
  {
    . . .
  }
}
```

НА ЗАМЕТКУ! До выпуска версии JDK 1.4 в механизме печати Java использовались диалоговые окна настройки печати, ориентированные именно на ту платформу, на которой работал пользователь. Для отображения платформенно-ориентированного диалогового окна настройки печати следует вызвать метод **`printDialog()`** без указания параметров. (В этом случае пользовательские установки нельзя объединить в набор атрибутов печати.)

Во время печати в методе `print()` из класса `PrinterJob` неоднократно вызывается метод `print()` для объекта типа `Printable`, связанного с текущим заданием на печать. Но поскольку в задании на печать неизвестно количество выводимых страниц, метод `print()` вызывается повторно, если он возвращает значение константы `Printable.PAGE_EXISTS`. Печать завершится лишь тогда, когда метод `print()` возвратит значение константы `Printable.NO_SUCH_PAGE`.

ВНИМАНИЕ! Отсчет номеров страниц, передаваемых методу **`print()`**, начинается с нуля.

Таким образом, задание на печать не может определить количество страниц до тех пор, пока печать не будет завершена. Поэтому в диалоговом окне настройки печати не может быть представлен фактический диапазон номеров страниц, а вместо него выводится сообщение `"Pages 1 to 1"` (Страниц от 1 до 1). В следующем разделе будет показано, как этот недостаток устраняется с помощью объекта типа `Book`, предоставляемого для задания на печать.

Итак, метод `print()` может повторно вызываться для объекта `Printable` в задании на печать, чтобы напечатать *одну и ту же* страницу. Поэтому вместо

подсчета количества страниц в методе `print()` лучше полагаться на номер страницы. Возможность многократного обращения к одной и той же странице предназначена для обслуживания некоторых печатающих устройств, особенно матричных и струйных, которые применяют *полосовой* способ печати. Они печатают сначала одну полосу, продвигают бумагу и затем печатают следующую полосу. Полосовой способ печати может использоваться даже на лазерных принтерах, которые обычно печатают сразу всю страницу. Это дает возможность управлять размером файла буферизации печати.

Если для вывода полосы в задании на печать требуется объект типа `Printable`, то следует указать область отсечения графического контекста печатаемой полосой и вызвать метод `print()`. Таким образом, все операции рисования будут выполняться только на данной полосе, и только в пределах ее прямоугольной области будут воспроизводиться рисуемые графические элементы. Методу `print()` совсем не обязательно знать об этих действиях, при условии, что он *не* затрагивает область отсечения.

ВНИМАНИЕ! Объект типа **`Graphics`**, возвращаемый методом **`print()`**, отсекается также по полям печатной страницы. Хотя можно рисовать на полях страницы, если переместить область отсечения. Но в графическом контексте печатающего устройства необходимо строго соблюдать заданную область отсечения. Для дальнейшего ограничения области отсечения следует вызывать метод **`clip()`**, а не **`setClip()`**. Если же требуется временно удалить область отсечения, то в начале разрабатываемого метода **`print()`** придется вызвать метод **`getClip()`**, чтобы получить эту область для ее последующего восстановления.

Параметр типа `PageFormat`, устанавливаемый при вызове метода `print()`, содержит сведения о печатаемой странице. Методы `getWidth()` и `getHeight()` возвращают формат бумаги, измеряемый в *пунктах*. Один пункт равен 1/72 дюйма, а один дюйм составляет 25,4 миллиметра. Например, формат бумаги A4 приблизительно равен 595×842 пункта, или 210×297 мм, а формат бумаги US Letter — 612×792 пункта, или 215,9×279,4 мм.

Пункты являются общепринятой единицей измерения в полиграфической промышленности США. Эта же единица применяется по умолчанию для всех графических контекстов печати. В этом можно убедиться на примере программы, исходный код которой приведен в конце данного раздела. Эта программа печатает две строки текста, которые находятся на расстоянии 72 пунктов друг от друга. Запустите эту программу, измерьте расстояние между напечатанными строками и убедитесь в том, что оно равно в точности 1 дюйму, или 25,4 мм.

Методы `getWidth()` и `getHeight()` из класса `PageFormat` возвращают полную ширину и высоту страницы. Но печатать можно не на всей странице. Обычно пользователи самостоятельно задают размеры полей печатаемой страницы, но даже если они этого не сделают вручную, поля печатаемой страницы все равно будут определены автоматически. Дело в том, что печатающему устройству необходимо каким-то образом удерживать листы бумаги, на которых производится печать, и поэтому по краям остаются небольшие непечатаемые участки.

Методы `getImageableWidth()` и `getImageableHeight()` возвращают ширину и высоту области, доступной для печати. Но поля не обязательно должны быть симметричными, поэтому следует также знать координаты верхнего

левого угла области, доступной для печати, которая условно показана на рис. 11.61. Эти координаты можно получить с помощью методов `getImageableX()` и `getImageableY()`.

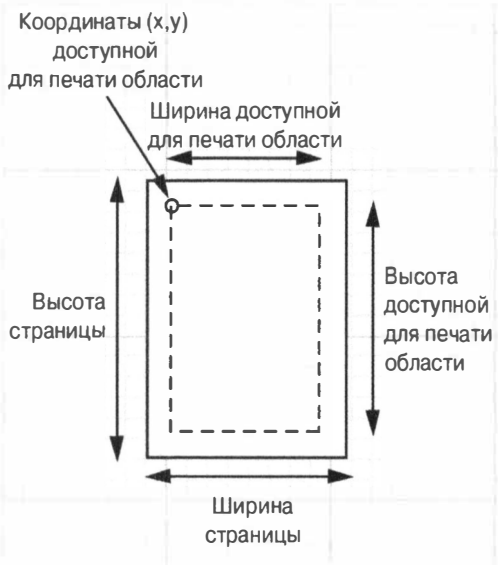


Рис. 11.61. Размеры страницы и области, доступной для печати

СОВЕТ. Графический контекст, который передается методу **`print()`**, обрезается по полям печатаемой страницы, тем не менее, начало системы координат остается по-прежнему в верхнем левом углу страницы. Поэтому начало координат имеет смысл перенести в левый верхний угол области, доступной для печати. Для этого разрабатываемый метод **`print()`** необходимо начать со следующей строки кода:

```
g.translate(pageFormat.getImageableX(), pageFormat.getImageableY());
```

Если же требуется предоставить пользователю возможность самостоятельно определять границы страницы и задавать книжную или альбомную ориентацию, не устанавливая других атрибутов печати, то для этого можно вызвать метод `pageDialog()` из класса `PrinterJob` следующим образом:

```
PageFormat format = job.pageDialog(attributes);
```

НА ЗАМЕТКУ! На одной из вкладок диалогового окна настройки печати содержатся параметры печатаемой страницы (рис. 11.62), которые пользователь может просмотреть и поправить непосредственно перед печатью. Это особенно удобно, если в программе реализован принцип WYSIWIG, когда на отпечатке получается именно то, что отображается на экране. Метод **`pageDialog()`** возвращает объект типа **`PageFormat`** со значениями параметров печати, задаваемых пользователем.

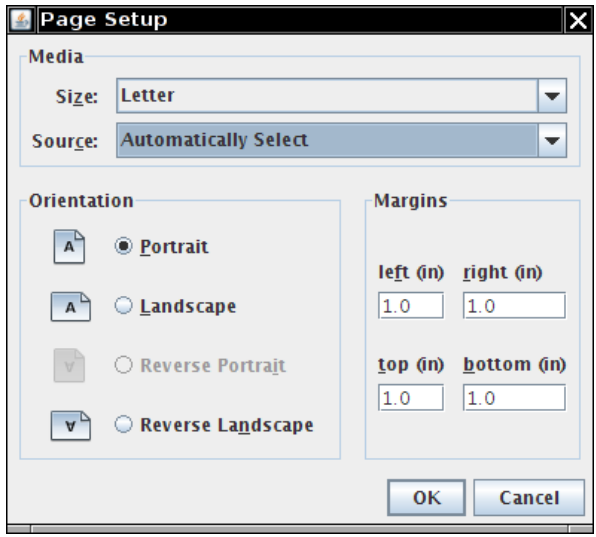


Рис. 11.62. Вид диалогового окна для настройки печатаемой страницы независимо от используемой платформы

В примере программы, исходный код которой приведен в листингах 11.23 и 11.24, показано, каким образом один и тот же ряд фигур выводится на экран и на печать. Класс `PrintPanel` расширяет класс `JPanel` и реализует интерфейс `Printable`, а его методы `paintComponent()` и `print()` вызывают один и тот же метод непосредственно для рисования, как выделено ниже полужирным.

```
class PrintPanel extends JPanel implements Printable
{
  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);
    var g2 = (Graphics2D) g;
    drawPage(g2);
  }

  public int print(Graphics g, PageFormat pf, int page)
        throws PrinterException
  {
    if (page >= 1) return Printable.NO_SUCH_PAGE;
    var g2 = (Graphics2D) g;
    g2.translate(pf.getImageableX(), pf.getImageableY());
    drawPage(g2);
    return Printable.PAGE_EXISTS;
  }
  public void drawPage(Graphics2D g2)
  {
    // здесь следует общий код для рисования
    . . .
  }
  . . .
}
```

В рассматриваемом здесь примере для демонстрации вывода данных на печать используется то же изображение, что и на рис. 11.50. В этом изображении контуры символов, составляющих строку "Hello, World", используются в качестве области отсечения для линий штрихового рисунка.

Для запуска задания на печать достаточно щелкнуть на экранной кнопке Print (Печать), а для открытия диалогового окна настройки печати — на кнопке Page setup (Параметры страницы). Соответствующий исходный код приведен в листинге 11.23.

НА ЗАМЕТКУ! Чтобы отобразить платформенно-ориентированное диалоговое окно параметров печатаемой страницы, методу **pageDialog()** следует передать объект типа **PageFormat**. Этот метод создает копию объекта, изменяет ее в соответствии с пользовательскими установками и возвращает обратно, как показано ниже.

```
PageFormat defaultFormat = printJob.defaultPage();
PageFormat selectedFormat =
    printJob.pageDialog(defaultFormat);
```

Листинг 11.23. Исходный код из файла **print/PrintTestFrame.java**

```
1  package print;
2
3  import java.awt.*;
4  import java.awt.print.*;
5
6  import javax.print.attribute.*;
7  import javax.swing.*;
8
9  /**
10  * В этом фрейме отображается панель с двухмерной
11  * графикой и экранными кнопками для печати графики
12  * и установки формата страницы
13  */
14 public class PrintTestFrame extends JFrame
15 {
16   private PrintComponent canvas;
17   private PrintRequestAttributeSet attributes;
18
19   public PrintTestFrame()
20   {
21     canvas = new PrintComponent();
22     add(canvas, BorderLayout.CENTER);
23
24     attributes = new HashPrintRequestAttributeSet();
25
26     var buttonPanel = new JPanel();
27     var printButton = new JButton("Print");
28     buttonPanel.add(printButton);
29     printButton.addActionListener(event ->
30       {
31         try
32         {
```

```
33            PrinterJob job = PrinterJob.getPrinterJob();
34            job.setPrintable(canvas);
35            if (job.printDialog(attributes))
36              job.print(attributes);
37          }
38          catch (PrinterException ex)
39          {
40            JOptionPane.showMessageDialog(
41                        PrintTestFrame.this, ex);
42          }
43        });
44
45      var pageSetupButton = new JButton("Page setup");
46      buttonPanel.add(pageSetupButton);
47      pageSetupButton.addActionListener(event ->
48        {
49          PrinterJob job = PrinterJob.getPrinterJob();
50          job.pageDialog(attributes);
51        });
52
53      add(buttonPanel, BorderLayout.NORTH);
54      pack();
55    }
56 }
```

Листинг 11.24. Исходный код из файла **print/PrintComponent.java**

```
1  package print;
2
3  import java.awt.*;
4  import java.awt.font.*;
5  import java.awt.geom.*;
6  import java.awt.print.*;
7  import javax.swing.*;
8
9  /**
10  * Этот компонент формирует двухмерную графику
11  * для вывода на экран и на печать
12  */
13 public class PrintComponent extends JComponent
14       implements Printable
15 {
16   private static final Dimension PREFERRED_SIZE =
17                        new Dimension(300, 300);
18   public void paintComponent(Graphics g)
19   {
20     var g2 = (Graphics2D) g;
21     drawPage(g2);
22   }
23
24   public int print(Graphics g, PageFormat pf, int page)
25         throws PrinterException
26   {
27     if (page >= 1) return Printable.NO_SUCH_PAGE;
```

```
28      var g2 = (Graphics2D) g;
29      g2.translate(pf.getImageableX(), pf.getImageableY());
30      g2.draw(new Rectangle2D.Double(0, 0,
31                                pf.getImageableWidth(),
32                                pf.getImageableHeight()));
33
34      drawPage(g2);
35      return Printable.PAGE_EXISTS;
36    }
37
38    /**
39     * Этот метод рисует страницу в графическом
40     * контексте экрана и печатающего устройства
41     * @param g2 Графический контекст
42     */
43    public void drawPage(Graphics2D g2)
44    {
45      FontRenderContext context =
46                        g2.getFontRenderContext();
47      var f = new Font("Serif", Font.PLAIN, 72);
48      var clipShape = new GeneralPath();
49
50      var layout = new TextLayout("Hello", f, context);
51      AffineTransform transform =
52            AffineTransform.getTranslateInstance(0, 72);
53      Shape outline = layout.getOutline(transform);
54      clipShape.append(outline, false);
55
56      layout = new TextLayout("World", f, context);
57      transform =
58           AffineTransform.getTranslateInstance(0, 144);
59      outline = layout.getOutline(transform);
60      clipShape.append(outline, false);
61
62      g2.draw(clipShape);
63      g2.clip(clipShape);
64
65      final int NLINES = 50;
66      var p = new Point2D.Double(0, 0);
67      for (int i = 0; i < NLINES; i++)
68      {
69        double x = (2 * getWidth() * i) / NLINES;
70        double y =
71           (2 * getHeight() * (NLINES - 1 - i)) / NLINES;
72        var q = new Point2D.Double(x, y);
73        g2.draw(new Line2D.Double(p, q));
74      }
75    }
76
77    public Dimension getPreferredSize()
78    { return PREFERRED_SIZE; }
79  }
```

java.awt.print.Printable 1.2

- **int print(Graphics *g*, PageFormat *format*, int *pageNumber*)**

  Воспроизводит *страницу и возвращает значение константы PAGE_EXISTS или NO_SUCH_PAGE.*

Параметры:	g	Графический контекст, в котором воспроизводится страница
	format	Формат воспроизводимой страницы
	pageNumber	Номер требуемой страницы

java.awt.print.PrinterJob 1.2

- **static PrinterJob getPrinterJob()**

  Возвращает объект задания на печать.

- **PageFormat defaultPage()**

  Возвращает установленный по умолчанию формат страницы для данного печатающего устройства.

- **boolean printDialog(PrintRequestAttributeSet *attributes*)**
- **boolean printDialog()**

  Открывают диалоговое окно, в котором пользователь может выбирать страницы для печати и настраивать ее параметры. Первый метод отображает платформенно-независимое диалоговое окно, а второй — платформенно-ориентированное окно. Оба метода изменяют заданный объект ***attributes*** с учетом пользовательских настроек и возвращают логическое значение **true**, если пользователь подтвердит установленные параметры настройки печати.

- **PageFormat pageDialog(PrintRequestAttributeSet *attributes*)**
- **PageFormat pageDialog(PageFormat *defaults*)**

  Выводят диалоговое окно настройки страницы. Первый из этих методов отображает платформенно-независимое диалоговое окно, а второй — платформенно-ориентированное окно. Оба метода возвращают объект типа **PageFormat**, определяющий формат, указанный пользователем в диалоговом окне. Первый метод видоизменяет заданный объект ***attributes*** таким образом, чтобы тот отражал пользовательские установки. Второй метод не видоизменяет объект ***defaults***.

- **void setPrintable(Printable *p*)**
- **void setPrintable(Printable *p*, PageFormat *format*)**

  Устанавливают объект типа **Printable** и необязательный формат страницы для текущего задания на печать.

- **void print()**
- **void print(PrintRequestAttributeSet *attributes*)**

  Выводят на печать текущий объект типа **Printable**, повторно вызывая его метод **print()** и отправляя воспроизводимые страницы на печатающее устройство до тех пор, пока не будут напечатаны все страницы.

java.awt.print.PageFormat 1.2

- **double getWidth()**
- **double getHeight()**

  Возвращают ширину и высоту страницы.
- **double getImageableWidth()**
- **double getImageableHeight()**

  Возвращают ширину и высоту доступной для печати области страницы.
- **double getImageableX()**
- **double getImageableY()**

  Возвращают координаты верхнего левого угла доступной для печати области страницы.
- **int getOrientation()**

  Возвращают значение одной из следующих констант, определяющих ориентацию страницы: **PORTRAIT**, **LANDSCAPE** или **REVERSE_LANDSCAPE**. Ориентация прозрачна для программирования, поскольку она автоматически учитывается в установках формата страниц и графического контекста.

11.5.2. Многостраничная печать

На практике объекты типа `Printable` нежелательно передавать заданию на печать без предварительной обработки. Сначала следует получить экземпляр класса, реализующего интерфейс `Pageable`. На платформе Java для этой цели предоставляется класс `Book`, реализующий книгу, состоящую из разделов, представленных объектами типа `Printable`. Для составления книги сначала вводятся объекты типа `Printable` в ее разделы и организуется нумерация страниц:

```
var book = new Book();
Printable coverPage = . . .;
Printable bodyPages = . . .;
// присоединить одну страницу:
book.append(coverPage, pageFormat);
book.append(bodyPages, pageFormat, pageCount);
```

Затем вызывается метод `setPageable()`, чтобы передать книгу в виде объекта типа `Book` заданию на печать:

```
printJob.setPageable(book);
```

Теперь заданию на печать точно известно количество печатаемых страниц, а в диалоговом окне настройки печати отображаются их номера, чтобы пользователь мог выбрать весь диапазон печатаемых страниц или только его часть.

ВНИМАНИЕ! Когда из задания на печать вызывается метод **print()** для объектов разделов типа **Printable**, ему передается текущий номер страницы, отсчитываемый в пределах *книги*, а не каждого *раздела*. Это очень неудобно, поскольку в каждом разделе должны храниться данные о количестве страниц в предыдущих разделах.

Для программирования наибольшую трудность при использовании класса `Book` представляет подсчет и сохранение данных о количестве страниц в каждом разделе во время печати. В объекте типа `Printable` следует составить *алгоритм*

компоновки печатаемой страницы. Компоновку страницы необходимо рассчитать перед началом ее печати, чтобы определить количество печатаемых страниц и их разрывы. Сведения о компоновке страницы можно сохранить для дальнейшего применения во время печати. Следует также учитывать, что пользователь может изменить формат страницы. В таком случае придется повторно рассчитать компоновку страницы, даже если печатаемые данные не изменились.

В примере программы из листинга 11.26 показано, каким образом организуется многостраничная печать. Эта программа выводит крупный заголовок (так называемый *баннер*), состоящий из строки символов очень большого размера, и поэтому они располагаются на нескольких страницах (рис. 11.63). После распечатки всех страниц поля обрезаются, а страницы склеиваются вместе.

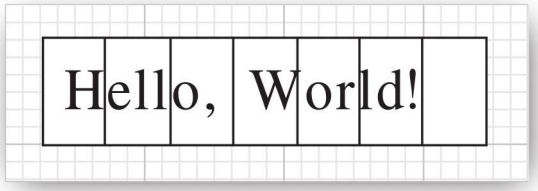


Рис. 11.63. Баннер

Метод `layoutPages()` из класса `Banner` рассчитывает компоновку страницы. Сначала компонуется строка сообщения, отформатированная шрифтом размером 72 пункта. Затем высота результирующей строки вычисляется и сравнивается с высотой печатаемой области страницы. На основании этих двух измерений определяется масштабный коэффициент. При выводе на печать строка увеличивается на этот масштабный коэффициент.

ВНИМАНИЕ! Для точного размещения на странице печатаемой информации требуется доступ к графическому контексту печатающего устройства. К сожалению, получить доступ к этому графическому контексту нельзя до тех пор, пока не начнется сама печать. В программе, рассматриваемой здесь в качестве примера, все требующиеся действия выполняются в экранном графическом контексте в надежде, что типографские параметры шрифтов на экране и на печатной странице совпадут.

В методе `getPageCount()` из класса `Banner` сначала вызывается метод определения компоновки страницы. Затем ширина символьной строки увеличивается в масштабе и делится на ширину доступной для печати области отдельной страницы. Частное от этого деления, округленное до следующего целого значения, определяет искомое количество страниц.

Из всего сказанного выше можно сделать вывод, что напечатать баннер не так-то просто, поскольку отдельные части символа могут располагаться на разных страницах. Но благодаря эффективным средствам прикладного интерфейса Java 2D API эта задача решается очень просто. Для получения конкретной страницы вызывается метод `translate()` из класса `Graphics2D`, смещающий верхний левый угол печатаемой символьной строки на заданное расстояние влево. Затем устанавливается прямоугольная область отсечения с размерами текущей страницы (рис. 11.64). Наконец, масштаб графического контекста изменяется по масштабному коэффициенту, вычисленному методом компоновки страницы.

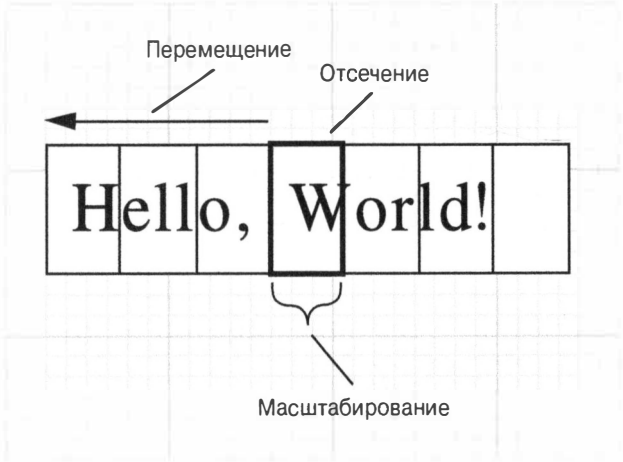


Рис. 11.64. Алгоритм постраничной печати баннера

Рассматриваемый здесь пример программы из листингов 25–28 демонстрирует истинный потенциал преобразований, благодаря которым код для воспроизведения графического изображения на странице остается простым. Как видите, преобразования выполняют всю работу по размещению изображения в соответствующем месте и отсечению той его части, которая выходит за пределы доступной для печати области страницы. В данном примере программы демонстрируется еще один наглядный пример применения преобразований для организации предварительного просмотра печатаемой страницы.

Листинг 11.25. Исходный код из файла **Book/BookTestFrame.java**

```
1  package book;
2
3  import java.awt.*;
4  import java.awt.print.*;
5
6  import javax.print.attribute.*;
7  import javax.swing.*;
8
9  /**
10  * Этот фрейм содержит текстовое поле для ввода
11  * баннера, а также кнопки для печати, настройки
12  * и предварительного просмотра печатаемой страницы
13  */
14 public class BookTestFrame extends JFrame
15 {
16   private JTextField text;
17   private PageFormat pageFormat;
18   private PrintRequestAttributeSet attributes;
19
20   public BookTestFrame()
21   {
22     text = new JTextField();
```

```
23      add(text, BorderLayout.NORTH);
24
25      attributes = new HashPrintRequestAttributeSet();
26
27      var buttonPanel = new JPanel();
28
29      var printButton = new JButton("Print");
30      buttonPanel.add(printButton);
31      printButton.addActionListener(event ->
32        {
33          try
34          {
35            PrinterJob job = PrinterJob.getPrinterJob();
36            job.setPageable(makeBook());
37            if (job.printDialog(attributes))
38            {
39              job.print(attributes);
40            }
41          }
42          catch (PrinterException e)
43          {
44            JOptionPane.showMessageDialog(
45                        BookTestFrame.this, e);
46          }
47        });
48
49      var pageSetupButton = new JButton("Page setup");
50      buttonPanel.add(pageSetupButton);
51      pageSetupButton.addActionListener(event ->
52        {
53          PrinterJob job = PrinterJob.getPrinterJob();
54          pageFormat = job.pageDialog(attributes);
55        });
56
57      var printPreviewButton =
58          new JButton("Print preview");
59      buttonPanel.add(printPreviewButton);
60      printPreviewButton.addActionListener(event ->
61        {
62          var dialog = new PrintPreviewDialog(makeBook());
63          dialog.setVisible(true);
64        });
65
66      add(buttonPanel, BorderLayout.SOUTH);
67      pack();
68    }
69
70    /**
71     * Составляет книгу из страницы
72     * обложки и страниц баннера
73     */
74    public Book makeBook()
75    {
76      if (pageFormat == null)
77      {
78        PrinterJob job = PrinterJob.getPrinterJob();
```

```
79        pageFormat = job.defaultPage();
80      }
81      var book = new Book();
82      String message = text.getText();
83      var banner = new Banner(message);
84      int pageCount = banner.getPageCount(
85          (Graphics2D) getGraphics(), pageFormat);
86      book.append(new CoverPage(message
87          + " (" + pageCount + " pages)"), pageFormat);
88      book.append(banner, pageFormat, pageCount);
89      return book;
90    }
91 }
```

Листинг 11.26. Исходный код из файла **book/Banner.java**

```
1  package book;
2
3  import java.awt.*;
4  import java.awt.font.*;
5  import java.awt.geom.*;
6  import java.awt.print.*;
7
8  /**
9   * Баннер для печати текстовой строки
10  * на нескольких страницах
11  */
12 public class Banner implements Printable
13 {
14   private String message;
15   private double scale;
16
17   /**
18    * Конструирует баннер
19    * @param m Строка сообщения
20   */
21   public Banner(String m)
22   {
23     message = m;
24   }
25
26   /**
27    * Получает количество страниц в данном разделе
28    * @param g2 Графический контекст
29    * @param pf Формат страницы
30    * @return Количество требующихся страниц
31   */
32   public int getPageCount(Graphics2D g2, PageFormat pf)
33   {
34     if (message.equals("")) return 0;
35     FontRenderContext context =
36                       g2.getFontRenderContext();
37     var f = new Font("Serif", Font.PLAIN, 72);
38     Rectangle2D bounds =
39                 f.getStringBounds(message, context);
```

```
40      scale = pf.getImageableHeight()
41              / bounds.getHeight();
42      double width = scale * bounds.getWidth();
43      int pages = (int) Math.ceil(
44                        width / pf.getImageableWidth());
45      return pages;
46    }
47
48    public int print(Graphics g, PageFormat pf, int page)
49          throws PrinterException
50    {
51      var g2 = (Graphics2D) g;
52      if (page > getPageCount(g2, pf))
53        return Printable.NO_SUCH_PAGE;
54      g2.translate(pf.getImageableX(),
55                   pf.getImageableY());
56
57      drawPage(g2, pf, page);
58      return Printable.PAGE_EXISTS;
59    }
60
61    public void drawPage(Graphics2D g2,
62                         PageFormat pf, int page)
63    {
64      if (message.equals("")) return;
65      page--; // учитывать страницу обложки
66
67      drawCropMarks(g2, pf);
68      g2.clip(new Rectangle2D.Double(0, 0,
69                              pf.getImageableWidth(),
70                              pf.getImageableHeight()));
71      g2.translate(-page * pf.getImageableWidth(), 0);
72      g2.scale(scale, scale);
73      FontRenderContext context =
74                        g2.getFontRenderContext();
75      var f = new Font("Serif", Font.PLAIN, 72);
76      var layout = new TextLayout(message, f, context);
77      AffineTransform transform = AffineTransform
78            .getTranslateInstance(0, layout.getAscent());
79      Shape outline = layout.getOutline(transform);
80      g2.draw(outline);
81    }
82
83    /**
84     * Рисует полудюймовые метки обрезки в углах страницы
85     * @param g2 Графический контекст
86     * @param pf Формат страницы
87    */
88    public void drawCropMarks(
89                Graphics2D g2, PageFormat pf)
90    {
91      // длина метки обрезки = 1/2 дюйма:
92      final double C = 36;
93      double w = pf.getImageableWidth();
94      double h = pf.getImageableHeight();
95      g2.draw(new Line2D.Double(0, 0, 0, C));
```

```
96      g2.draw(new Line2D.Double(0, 0, C, 0));
97      g2.draw(new Line2D.Double(w, 0, w, C));
98      g2.draw(new Line2D.Double(w, 0, w - C, 0));
99      g2.draw(new Line2D.Double(0, h, 0, h - C));
100     g2.draw(new Line2D.Double(0, h, C, h));
101     g2.draw(new Line2D.Double(w, h, w, h - C));
102     g2.draw(new Line2D.Double(w, h, w - C, h));
103   }
104 }
105
106 /**
107  * Этот класс печатает страницу обложки с заглавием
108  */
109 class CoverPage implements Printable
110 {
111   private String title;
112
113   /**
114    * Конструирует страницу обложки
115    * @param t Заглавие
116   */
117   public CoverPage(String t)
118   {
119     title = t;
120   }
121
122   public int print(Graphics g, PageFormat pf, int page)
123         throws PrinterException
124   {
125     if (page >= 1) return Printable.NO_SUCH_PAGE;
126     var g2 = (Graphics2D) g;
127     g2.setPaint(Color.black);
128     g2.translate(pf.getImageableX(),
129                  pf.getImageableY());
130     FontRenderContext context =
131                       g2.getFontRenderContext();
132     Font f = g2.getFont();
133     var layout = new TextLayout(title, f, context);
134     float ascent = layout.getAscent();
135     g2.drawString(title, 0, ascent);
136     return Printable.PAGE_EXISTS;
137   }
138 }
```

Листинг 11.27. Исходный код из файла **book/PrintPreviewDialog.java**

```
1  package book;
2
3  import java.awt.*;
4  import java.awt.print.*;
5
6  import javax.swing.*;
7
8  /**
9   * Этот класс реализует типичное окно
```

```
10  * предварительного просмотра печати
11  */
12 public class PrintPreviewDialog extends JDialog
13 {
14   private static final int DEFAULT_WIDTH = 300;
15   private static final int DEFAULT_HEIGHT = 300;
16
17   private PrintPreviewCanvas canvas;
18
19   /**
20    * Конструирует диалоговое окно
21    * предварительного просмотра печати
22    * @param p Печатаемый объект типа Printable
23    * @param pf Формат страницы
24    * @param pages Количество страниц в
25    *              печатаемом объекте p
26    */
27   public PrintPreviewDialog(
28         Printable p, PageFormat pf, int pages)
29   {
30     var book = new Book();
31     book.append(p, pf, pages);
32     layoutUI(book);
33   }
34
35   /**
36    * Создает диалоговое окно предварительного
37    * просмотра печати
38    * @param b Объект книги типа Book
39    */
40   public PrintPreviewDialog(Book b)
41   {
42     layoutUI(b);
43   }
44
45   /**
46    * Компонует ГПИ в диалоговом окне
47    * @param book Предварительно просматриваемая книга
48    */
49   public void layoutUI(Book book)
50   {
51     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
52
53     canvas = new PrintPreviewCanvas(book);
54     add(canvas, BorderLayout.CENTER);
55
56     var buttonPanel = new JPanel();
57
58     var nextButton = new JButton("Next");
59     buttonPanel.add(nextButton);
60     nextButton.addActionListener(event ->
61                canvas.flipPage(1));
62
63     var previousButton = new JButton("Previous");
64     buttonPanel.add(previousButton);
65     previousButton.addActionListener(event ->
```

```
66                    canvas.flipPage(-1));
67
68     var closeButton = new JButton("Close");
69     buttonPanel.add(closeButton);
70     closeButton.addActionListener(event ->
71                 setVisible(false));
72
73     add(buttonPanel, BorderLayout.SOUTH);
74   }
75 }
```

Листинг 11.28. Исходный код из файла **book/PrintPreviewCanvas.java**

```
1  package book;
2
3  import java.awt.*;
4  import java.awt.geom.*;
5  import java.awt.print.*;
6  import javax.swing.*;
7
8  /**
9   * Холст для отображения предварительного
10  * просмотра печати
11  */
12 class PrintPreviewCanvas extends JComponent
13 {
14   private Book book;
15   private int currentPage;
16
17   /**
18    * Конструирует холст для предварительного
19    * просмотра печати
20    * @param b Предварительно просматриваемая книга
21   */
22   public PrintPreviewCanvas(Book b)
23   {
24     book = b;
25     currentPage = 0;
26   }
27   public void paintComponent(Graphics g)
28   {
29     var g2 = (Graphics2D) g;
30     PageFormat pageFormat =
31     book.getPageFormat(currentPage);
32     // координата x смещения начала страницы в окне:
33     double xoff;
34     // координата y смещения начала страницы в окне:
35     double yoff;
36     // масштабный коэффициент для подгонки
37     // просматриваемой страницы по размерам окна:
38     double scale;
39     double px = pageFormat.getWidth();
40     double py = pageFormat.getHeight();
41     double sx = getWidth() - 1;
42     double sy = getHeight() - 1;
```

```
43      // отцентровать по горизонтали:
44      if (px / py < sx / sy)
45      {
46        scale = sy / py;
47        xoff = 0.5 * (sx - scale * px);
48        yoff = 0;
49      }
50      else
51      // отцентровать по вертикали:
52      {
53        scale = sx / px;
54        xoff = 0;
55        yoff = 0.5 * (sy - scale * py);
56      }
57      g2.translate((float) xoff, (float) yoff);
58      g2.scale((float) scale, (float) scale);
59
60      // нарисовать контуры страницы page, игнорируя поля:
61      var page = new Rectangle2D.Double(0, 0, px, py);
62      g2.setPaint(Color.white);
63      g2.fill(page);
64      g2.setPaint(Color.black);
65      g2.draw(page);
66
67      Printable printable =
68                book.getPrintable(currentPage);
69      try
70      {
71        printable.print(g2, pageFormat, currentPage);
72      }
73      catch (PrinterException e)
74      {
75        g2.draw(new Line2D.Double(0, 0, px, py));
76        g2.draw(new Line2D.Double(px, 0, 0, py));
77      }
78    }
79    /**
80     * Листать книгу на заданное число страниц
81     * @param by Число листаемых страниц.
82     *           Отрицательные значения данного
83     *           параметра обозначают листание книги назад
84    */
85    public void flipPage(int by)
86    {
87      int newPage = currentPage + by;
88      if (0 <= newPage
89                && newPage < book.getNumberOfPages())
90      {
91        currentPage = newPage;
92         repaint();
93      }
94    }
95  }
```

11.5.3. Службы печати

В предыдущих разделах было показано, как выводить двухмерную графику на печать. Но прикладной интерфейс API для печати, внедренный в версии Java 1.4, обладает более широкими возможностями. В этом прикладном интерфейсе определен целый ряд типов данных и поддерживается поиск служб печати для их вывода. Ниже перечислены некоторые из поддерживаемых типов данных.

- Изображения в формате GIF, JPEG или PNG.
- Текстовые документы в формате HTML, PostScript или PDF.
- Неформатированные данные для печатающего устройства.
- Экземпляры классов, реализующих интерфейсы `Printable`, `Pageable` или `RenderableImage`.

Сами данные могут храниться в любом источнике байтов или символов, например, в потоке ввода, веб-ресурсе, доступном по указанному URL, или в массиве. Сочетание типа данных и источника данных называется *разновидностью документа*. Так, в классе `DocFlavor` определен целый ряд внутренних классов для различных источников данных. Каждый из этих внутренних классов содержит константы для указания разновидности документа. Например, приведенная ниже константа описывает изображение формата GIF, считываемое из потока ввода. Все разновидности документов, т.е. допустимые сочетания типов источников и данных, включая и тип MIME, перечислены в табл. 11.4.

```
DocFlavor.INPUT_STREAM.GI
```

Таблица 11.4. Разновидности документов для служб печати

Источник данных	Тип данных	Тип MIME
`INPUT_STREAM`	`GIF`	`image/gif`
`URL`	`JPEG`	`image/jpeg`
`BYTE_ARRAY`	`PNG`	`image/png`
	`POSTSCRIPT`	`application/postscript`
	`PDF`	`application/pdf`
	`TEXT_HTML_HOST`	`text/html` (в кодировке, характерной для конкретного хоста)
	`TEXT_HTML_US_ASCII`	`text/html; charset=us-ascii`
	`TEXT_HTML_UTF_8`	`text/html; charset=utf-8`
	`TEXT_HTML_UTF_16`	`text/html; charset=utf-16`
	`TEXT_HTML_UTF_16LE`	`text/html; charset=utf-16le` (прямой порядок следования байтов)
	`TEXT_HTML_UTF_16BE`	`text/html; charset=utf-16be` (обратный порядок следования байтов)
	`TEXT_PLAIN_HOST`	`text/plain` (в кодировке, характерной для конкретного хоста)
	`TEXT_PLAIN_US_ASCII`	`text/plain; charset=us-ascii`
	`TEXT_PLAIN_UTF_8`	`text/plain; charset=utf-8`

Окончание табл. 11.4

Источник данных	Тип данных	Тип MIME
	TEXT_PLAIN_UTF_16	**text/plain; charset=utf-16**
	TEXT_PLAIN_UTF_16LE	**text/plain; charset=utf-16le** (прямой порядок следования байтов)
	TEXT_PLAIN_UTF_16BE	**text/plain; charset=utf-16be** (обратный порядок следования байтов)
	PCL	**application/vnd.hp-PCL** (Hewlett Packard Printer Control Language — язык управления печатью от компании Hewlett)
	AUTOSENSE	**application/octet-stream** (неформатированные данные для печатающего устройства)
READER	**TEXT_HTML**	**text/html; charset=utf-16**
STRING	**TEXT_PLAIN**	**text/plain; charset=utf-16**
CHAR_ARRAY		
SERVICE_FORMATTED	**PRINTABLE**	Отсутствует
	PAGEABLE	Отсутствует
	RENDERABLE_IMAGE	Отсутствует

Допустим, требуется напечатать растровое изображение из файла формата GIF. Для этого необходимо сначала выяснить, имеется ли соответствующая *служба печати*, способная справиться с подобной задачей. Статический метод `lookupPrintServices()` из класса `PrintServiceLookup` возвращает массив объектов типа `PrintService`, способных напечатать указанную разновидность документа, как показано ниже.

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;
PrintService[] services = PrintServiceLookup
              .lookupPrintServices(flavor, null);
```

Второй параметр метода `lookupPrintServices()` принимает пустое значение `null`, которое означает, что поиск подходящей службы печати не ограничивается какими-то определенными атрибутами печати. Более подробно атрибуты печати рассматриваются в следующем разделе.

Если в результате поиска служб печати получен массив типа **`PrintService[]`** с несколькими элементами, то среди перечисленных в нем служб печати следует выбирать какую-нибудь одну. Чтобы извлечь список имен служб печати (например, имен печатающих устройств) и предоставить пользователю возможность выбора, достаточно вызвать метод **`getName()`** из класса **`PrintService`**.

Далее из выбранной службы следует получить задание на печать:

```
DocPrintJob job = services[i].createPrintJob();
```

Для печати документа определенной разновидности необходимо создать объект, класс которого реализует интерфейс **`Doc`**. Для этой цели в библиотеке Java предусмотрен класс `SimpleDoc`. Конструктору класса `SimpleDoc` следует передать объект, представляющий источник данных, разновидность документа и дополнительный, но необязательный набор атрибутов печати, как показано в приведенном ниже примере кода.

```
var in = new FileInputStream(fileName);
var doc = new SimpleDoc(in, flavor, null);
```

Наконец, когда все будет готово к печати, можно приступить к выводу документа на печать, вызвав приведенный ниже метод. Как и прежде, пустое значение `null` второго параметра этого метода можно заменить набором атрибутов печати.

```
job.print(doc, null);
```

Однако описанный выше процесс печати совершенно не похож на рассматривавшийся в предыдущем разделе, потому что в данном случае отсутствует всякое взаимодействие с пользователем посредством диалоговых окон. Применяя подобный механизм, можно организовать печать на сервере, где пользователи должны передавать задания на печать из специальной веб-формы.

javax.print.PrintServiceLookup 1.4

- **PrintService[] lookupPrintServices(DocFlavor *flavor*, AttributeSet *attributes*)**
  Ищет службы печати, способные справиться с указанной разновидностью документа и атрибутами печати.

Параметры:	***flavor***	Разновидность документа
	attributes	Требуемые атрибуты печати или пустое значение **null**, если эти атрибуты не принимаются во внимание

javax.print.PrintService 1.4

- **DocPrintJob createPrintJob()**
  Составляет задание на печать для вывода экземпляра класса, реализующего интерфейс **Doc** (например, **SimpleDoc**).

javax.print.DocPrintJob 1.4

- **void print(Doc *doc*, PrintRequestAttributeSet *attributes*)**
  Выводит на печать заданный документ с указанными атрибутами печати.

Параметры:	***doc***	Печатаемый документ
	attributes	Требуемые атрибуты печати или пустое значение **null**, если эти атрибуты не принимаются во внимание

javax.print.SimpleDoc 1.4

- **SimpleDoc(Object *data*, DocFlavor *flavor*, DocAttributeSet *attributes*)**

  Создает объект типа **SimpleDoc**, который может быть выведен на печать в задании типа **DocPrintJob**.

  Параметры:

data	Объект с данными для печати, например, поток ввода или объект типа **Printable**
flavor	Разновидность документа с данными для печати
attributes	Атрибуты печати документа или пустое значение **null**, если атрибуты печати не требуются

11.5.4. Потоковые службы печати

Обычные службы печати направляют данные на печатающее устройство. Потоковые службы печати формируют данные аналогичным образом, но направляют их в поток вывода. Такая потребность возникает в тех случаях, когда необходимо задержать печать или интерпретировать формат печати данных в других программах. Если, например, данные печатаются в формате PostScript, их целесообразно сохранить в файле, поскольку многие программы способны обращаться с файлами формата PostScript. На платформе Java предусмотрена потоковая служба печати, способная распечатывать растровые изображения и двухмерную графику в формате PostScript. Эти службы можно использовать во всех системах — даже в тех, где нет локальных печатающих устройств.

Перечисление потоковых служб печати организовано немного сложнее, чем локальных служб печати. Для этого необходимо получить сначала разновидность печатаемого документа в виде объекта типа `DocFlavor` и тип MIME потока вывода, а затем массив фабричных объектов, представляющих потоковые службы печати:

```
DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
String mimeType = "application/postscript";
StreamPrintServiceFactory[] factories =
      StreamPrintServiceFactory
      .lookupStreamPrintServiceFactories(flavor, mimeType);
```

В классе `StreamPrintServiceFactory` нет ни одного метода, с помощью которого можно было бы отличить один фабричный объект потоковой службы от другого, поэтому выбирается самый первый объект, находящийся в элементе массива `factories[0]`. Для получения объекта потоковой службы печати типа `StreamPrintService` следует вызвать приведенный ниже метод `getPrintService()`, указав поток вывода в качестве его параметра. Класс `StreamPrintService` является производным от класса `PrintService`, поэтому для вывода на печать остается лишь выполнить действия, описанные в предыдущем разделе.

```
var out = new FileOutputStream(fileName);
StreamPrintService service =
      factories[0].getPrintService(out);
```

javax.print.StreamPrintServiceFactory 1.4

- **StreamPrintServiceFactory[] lookupStreamPrintServiceFactories(DocFlavor *flavor*, String *mimeType*)**

  Ищет потоковые службы печати, способные напечатать документ указанной разновидности и создать поток вывода заданного типа MIME.

- **StreamPrintService getPrintService(OutputStream *out*)**

  Получает службу печати, направляющую данные в указанный поток вывода.

В примере программы из листинга 11.29 показывается, как пользоваться потоковой службой печати для вывода двухмерных фигур на печать в файл формата PostScript. Фрагмент кода для рисования фигур можно заменить в данной программе на фрагмент кода для формирования любых двухмерных фигур средствами прикладного интерфейса Java 2D API, чтобы вывести их на печать в формате PostScript. Полученный в итоге файл нетрудно затем преобразовать в формат PDF или EPS, используя внешнее инструментальное средство. (К сожалению, вывод на печать в форматах PDF и EPS непосредственно в Java не поддерживается.)

НА ЗАМЕТКУ! В данном примере метод **draw()** вызывается для рисования двухмерных фигур в графическом контексте объекта типа **Graphics2D**. Если же требуется нарисовать поверхность конкретного компонента (например, таблицы или дерева), с этой целью можно воспользоваться следующим фрагментом кода:

```
private static int IMAGE_WIDTH = component.getWidth();
private static int IMAGE_HEIGHT = component.getHeight();
public static void draw(Graphics2D g2)
{ component.paint(g2); }
```

Листинг 11.29. Исходный код из файла **printService/PrintServiceTest.java**

```
1  package printService;
2
3  import java.awt.*;
4  import java.awt.font.*;
5  import java.awt.geom.*;
6  import java.awt.print.*;
7  import java.io.*;
8  import javax.print.*;
9  import javax.print.attribute.*;
10
11 /**
12  * В этой программе демонстрируется применение
13  * потоковых служб печати. В ней двухмерные фигуры
14  * выводятся в файл формата PostScript. Если имя
15  * целевого файла не указано в командной строке,
16  * полученный результат сохраняется в файле out.ps
17  * @version 1.0 2018-06-01
18  * @author Cay Horstmann
19  */
20 public class PrintServiceTest
```

```
21  {
22    // здесь задаются размеры изображения:
23    private static int IMAGE_WIDTH = 300;
24    private static int IMAGE_HEIGHT = 300;
25
25    public static void draw(Graphics2D g2)
27    {
28      // ниже следуют инструкции для рисования:
29      FontRenderContext context =
30                        g2.getFontRenderContext();
31      var f = new Font("Serif", Font.PLAIN, 72);
32      var clipShape = new GeneralPath();
33
34      var layout = new TextLayout("Hello", f, context);
35      AffineTransform transform =
36            AffineTransform.getTranslateInstance(0, 72);
37      Shape outline = layout.getOutline(transform);
38      clipShape.append(outline, false);
39
40      layout = new TextLayout("World", f, context);
41      transform =
42           AffineTransform.getTranslateInstance(0, 144);
43      outline = layout.getOutline(transform);
44      clipShape.append(outline, false);
45
46      g2.draw(clipShape);
47      g2.clip(clipShape);
48
49      final int NLINES = 50;
50      var p = new Point2D.Double(0, 0);
51      for (int i = 0; i < NLINES; i++)
52      {
53        double x = (2 * IMAGE_WIDTH * i) / NLINES;
54        double y = (2 * IMAGE_HEIGHT * (NLINES - 1 - i))
55                        / NLINES;
56        var q = new Point2D.Double(x, y);
57        g2.draw(new Line2D.Double(p, q));
58      }
59    }
60
61    public static void main(String[] args)
62          throws IOException, PrintException
63    {
64      String fileName =
65            args.length > 0 ? args[0] : "out.ps";
66      DocFlavor flavor =
67               DocFlavor.SERVICE_FORMATTED.PRINTABLE;
68      var mimeType = "application/postscript";
69      StreamPrintServiceFactory[] factories =
70            StreamPrintServiceFactory
71            .lookupStreamPrintServiceFactories(
72                        flavor, mimeType);
73      var out = new FileOutputStream(fileName);
74      if (factories.length > 0)
```

```
75      {
76        PrintService service =
77                    factories[0].getPrintService(out);
78        var doc = new SimpleDoc(new Printable()
79          {
80            public int print(Graphics g,
81                             PageFormat pf, int page)
82            {
83              if (page >= 1)
84                return Printable.NO_SUCH_PAGE;
85              else
86              {
87                double sf1 = pf.getImageableWidth()
88                             / (IMAGE_WIDTH + 1);
89                double sf2 = pf.getImageableHeight()
90                             / (IMAGE_HEIGHT + 1);
91                double s = Math.min(sf1, sf2);
92                var g2 = (Graphics2D) g;
93                g2.translate(
94                    (pf.getWidth()
95                     - pf.getImageableWidth()) / 2,
96                    (pf.getHeight()
97                     - pf.getImageableHeight()) / 2);
98                g2.scale(s, s);
99                draw(g2);
100               return Printable.PAGE_EXISTS;
101             }
102           }
103         }, flavor, null);
104       DocPrintJob job = service.createPrintJob();
105       var attributes = new HashPrintRequestAttributeSet();
106       job.print(doc, attributes);
107     }
108     else
109       System.out.println("No factories for " + mimeType);
110   }
111 }
```

11.5.5. Атрибуты печати

Прикладной интерфейс API для служб печати содержит сложный набор интерфейсов и классов, предназначенных для указания различных типов атрибутов печати, которые делятся на четыре основные группы. Первые две группы атрибутов печати определяют запросы, направляемые на печатающее устройство.

- *Атрибуты запроса печати.* Запрашивают отдельные характеристики всех объектов документов в задании на печать. В качестве примеров таких характеристик можно привести двухстороннюю печать или формат бумаги.
- *Атрибуты документа.* Используются только для какого-нибудь одного объекта документа.

Остальные две группы атрибутов содержат сведения о печатающем устройстве и состоянии задания на печать.

- *Атрибуты службы печати*. Содержат сведения о службе печати (например, о производителе и модели печатающего устройства, а также о том, принимает ли печатающее устройство задание на печать в настоящий момент).
- *Атрибуты задания на печать*. Содержат сведения о состоянии определенного задания на печать (например, завершено ли задание или еще не выполнено).

Для описания различных атрибутов печати предусмотрен интерфейс `Attribute` и перечисленные ниже подчиненные интерфейсы.

```
PrintRequestAttribute
DocAttribute
PrintServiceAttribute
PrintJobAttribute
SupportedValuesAttribute
```

Отдельные классы атрибутов реализуют один или несколько этих интерфейсов. Например, экземпляры класса `Copies` используются для описания количества копий печатаемого документа. Этот класс реализует интерфейсы `PrintRequestAttribute` и `PrintJobAttribute`. Задание на печать может содержать запрос на печать нескольких копий, тогда как атрибутом задания на печать может быть количество фактически напечатанных копий. Фактическое количество копий может оказаться меньше требуемого из-за ограничений, накладываемых печатающим устройством, или исчерпания бумаги.

Интерфейс `SupportedValuesAttribute` указывает, что значения атрибута не отражают фактический запрос или текущее состояние, а демонстрируют функциональные возможности службы печати. Например, объект класса `CopiesSupported` реализует интерфейс `SupportedValuesAttribute` и сообщает, что печатающее устройство поддерживает печать от 1 до 99 копий. На рис. 11.65 схематически показана иерархия наследования интерфейсов и классов, реализующих атрибуты печати.

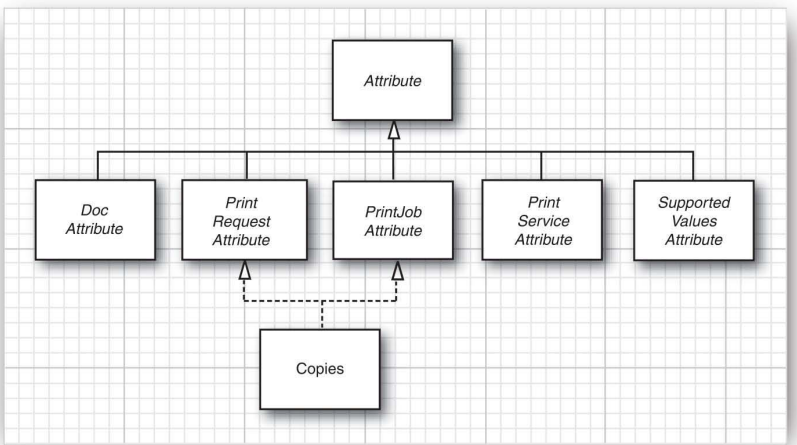


Рис. 11.65. Иерархия наследования интерфейсов и классов, реализующих атрибуты печати

Помимо интерфейсов и классов для отдельных атрибутов печати, в прикладном интерфейсе API для служб печати предусмотрены интерфейсы и классы для наборов атрибутов печати. Так, у интерфейса `AttributeSet` имеются следующие подчиненные интерфейсы:

```
PrintRequestAttributeSet
DocAttributeSet
PrintServiceAttributeSet
PrintJobAttributeSet
```

Для каждого из них предусмотрен класс, реализующий свой интерфейс. Всего насчитывается пять таких классов, перечисленных ниже. На рис. 11.66 схематически показана иерархия наследования интерфейсов и классов, реализующих наборы атрибутов печати.

```
HashAttributeSet
HashPrintRequestAttributeSet
HashDocAttributeSet
HashPrintServiceAttributeSet
HashPrintJobAttributeSet
```

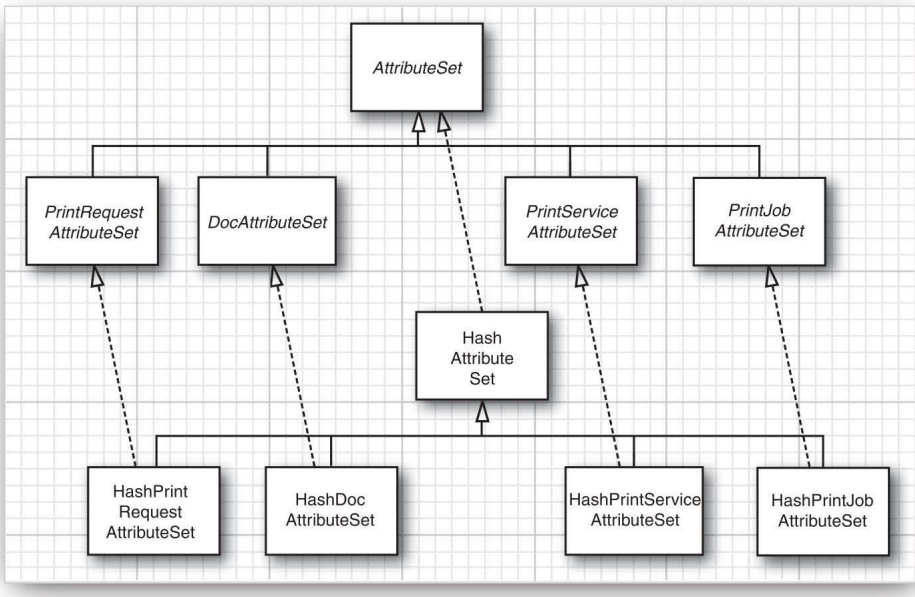


Рис. 11.66. Иерархия наследования интерфейсов и классов, реализующих наборы атрибутов печати

Например, создать набор атрибутов для запроса на печать можно следующим образом:

```
var attributes = new HashPrintRequestAttributeSet();
```

После создания набора атрибутов печати потребность в префиксе `Hash` отпадает. А зачем вообще нужны все эти интерфейсы? Они дают возможность проверить правильность использования атрибутов печати. Например, интерфейс

DocAttributeSet принимает только те объекты, классы которых реализуют интерфейс DocAttribute, а любая попытка добавить какой-нибудь другой атрибут печати приведет к ошибке.

Набор атрибутов печати представляет собой особый вид отображения, где ключи являются объектами класса Class, а значения — объектами класса, реализующего интерфейс Attribute. Так, если создать и вставить объект в набор атрибутов печати с помощью операции new Copies(10), то его ключом будет объект Copies.class типа Class. Этот ключ называется *категорией* атрибутов печати. В интерфейсе Attribute объявляется приведенный ниже метод, возвращающий категорию атрибута печати. Класс Copies содержит определение этого метода для возврата объекта Copies.class, но совсем не обязательно, чтобы категория была объектом того же класса, что и класс атрибута печати.

```
Class getCategory()
```

При вводе атрибута печати в набор атрибутов категория извлекается автоматически, поэтому достаточно ввести только значение атрибута печати, как показано ниже. Если же впоследствии ввести другой атрибут печати из той же самой категории, он будет записан вместо первого атрибута.

```
attributes.add(new Copies(10));
```

Категория используется в качестве ключа для извлечения атрибута печати, как показано ниже.

```
AttributeSet attributes = job.getAttributes();
var copies = (Copies) attribute.get(Copies.class);
```

Наконец, атрибуты печати группируются по типам значений. Например, атрибут типа Copies может иметь любое целочисленное значение, поскольку класс Copies является расширением класса IntegerSyntax, предназначенного для обработки всех целочисленных атрибутов печати. Метод getValue() из этого класса возвращает целочисленное значение атрибута печати:

```
int n = copies.getValue();
```

Перечисленные ниже классы инкапсулируют соответственно символьную строку, дату/время или URI.

```
TextSyntax
DateTimeSyntax
URISyntax
```

К тому же многие атрибуты печати могут принимать ограниченное количество значений. Например, у атрибута типа PrintQuality имеются всего три допустимых значения для указания экономного, нормального и высокого качества печати, представленных следующими константами:

```
PrintQuality.DRAFT
PrintQuality.NORMAL
PrintQuality.HIGH
```

Классы атрибутов печати с ограниченным количеством значений расширяют класс EnumSyntax, предоставляющий удобные методы для установки перечислимых значений с учетом типовой безопасности. Наличие таких классов избавляет от необходимости беспокоиться о конкретном механизме реализации при

использовании атрибутов печати. Достаточно ввести в набор атрибутов печати именованные значения следующим образом:

```
attributes.add(PrintQuality.HIGH);
```

Ниже показано, как проверить значение атрибута печати.

```
if (attributes.get(PrintQuality.class) == PrintQuality.HIGH)
   . . .
```

Атрибуты печати перечислены в табл. 11.5. Во втором столбце приведен суперкласс каждого класса атрибута (например, класс `IntegerSyntax` для атрибута типа `Copies`) или набор перечислимых значений. В последних четырех столбцах указаны интерфейсы, которые реализует данный класс атрибута: `DocAttribute` (DA), `PrintJobAttribute` (PJA), `PrintRequestAttribute` (PRA) и `PrintServiceAttribute` (PSA).

Таблица 11.5. Атрибуты печати

Атрибут	Суперкласс или набор констант перечислимого типа	DA	PJA	PRA	PSA
`Chromaticity`	`MONOCHROME`, `COLOR`	✓	✓	✓	
`ColorSupported`	`SUPPORTED`, `NOT_SUPPORTED`				✓
`Compression`	`COMPRESS`, `DEFLATE`, `GZIP`, `NONE`	✓			
`Copies`	`IntegerSyntax`		✓	✓	
`DateTimeAtCompleted`	`DateTimeSyntax`		✓		
`DateTimeAtCreation`	`DateTimeSyntax`		✓		
`DateTimeAtProcessing`	`DateTimeSyntax`		✓		
`Destination`	`URISyntax`		✓	✓	
`DocumentName`	`TextSyntax`	✓			
`Fidelity`	`FIDELITY_TRUE`, `FIDELITY_FALSE`		✓	✓	
`Finishings`	`NONE`, `STAPLE`, `EDGE_STITCH`, `BIND`, `SADDLE_STITCH`, `COVER`, . . .	✓	✓	✓	
`JobHoldUntil`	`DateTimeSyntax`		✓	✓	
`JobImpressions`	`IntegerSyntax`		✓	✓	
`JobImpressionsCompleted`	`IntegerSyntax`		✓		
`JobKOctets`	`IntegerSyntax`		✓	✓	
`JobKOctetsProcessed`	`IntegerSyntax`		✓		
`JobMediaSheets`	`IntegerSyntax`		✓	✓	
`JobMediaSheetsCompleted`	`IntegerSyntax`		✓		
`JobMessageFromOperator`	`TextSyntax`		✓		
`JobName`	`TextSyntax`		✓	✓	
`JobOriginatingUserName`	`TextSyntax`		✓		
`JobPriority`	`IntegerSyntax`		✓	✓	
`JobSheets`	`STANDARD`, `NONE`		✓	✓	
`JobState`	`ABORTED`, `CANCELED`, `COMPLETED`, `PENDING`, `PENDING_HELD`, `PROCESSING`, `PROCESSING_STOPPED`		✓		
`JobStateReason`	`ABORTED_BY_SYSTEM`, `DOCUMENT_FORMAT_ERROR` и проч.				

Продолжение табл. 10.5

Атрибут	Суперкласс или набор констант перечислимого типа	DA	PJA	PRA	PSA
JobStateReasons	HashSet		✓		
MediaName	ISO_A4_WHITE, ISO_A4_TRANSPARENT, NA_LETTER_WHITE, NA_LETTER_TRANSPARENT	✓	✓	✓	
MediaSize	ISO.A0–ISO.A10, ISO.B0–ISO.B10, ISO.C0–ISO.C10, NA.LETTER, NA.LEGAL и другие форматы страниц и бумаги				
MediaSizeName	ISO.A0–ISO.A10, ISO.B0–ISO.B10, ISO.C0–ISO.C10, NA.LETTER, NA.LEGAL и другие форматы страниц и бумаги	✓	✓	✓	
MediaTray	TOP, MIDDLE, BOTTOM, SIDE, ENVELOPE, LARGE_CAPACITY, MAIN, MANUAL	✓	✓	✓	
MultipleDocumentHandling	SINGLE_DOCUMENT, SINGLE_DOCUMENT_NEW_SHEET, SEPARATE_DOCUMENTS_COLLATED_COPIES, SEPARATE_DOCUMENTS_UNCOLLATED_COPIES		✓	✓	
NumberOfDocuments	IntegerSyntax		✓		
NumberOfInterveningJobs	IntegerSyntax		✓		
NumberUp	IntegerSyntax	✓	✓	✓	
OrientationRequested	PORTRAIT, LANDSCAPE, REVERSE_PORTRAIT, REVERSE_LANDSCAPE	✓	✓	✓	
OutputDeviceAssigned	TextSyntax		✓		
PageRanges	SetOfInteger	✓	✓	✓	
PagesPerMinute	IntegerSyntax				✓
PagesPerMinuteColor	IntegerSyntax				✓
PDLOverrideSupported	ATTEMPTED, NOT_ATTEMPTED				✓
PresentationDirection	TORIGHT_TOBOTTOM, TORIGHT_TOTOP, TOBOTTOM_TORIGHT, TOBOTTOM_TOLEFT, TOLEFT_TOBOTTOM, TOLEFT_TOTOP, TOTOP_TORIGHT, TOTOP_TOLEFT		✓	✓	
PrinterInfo	TextSyntax				✓
PrinterIsAcceptingJobs	ACCEPTING_JOBS, NOT_ACCEPTING_JOBS				✓
PrinterLocation	TextSyntax				✓
PrinterMakeAndModel	TextSyntax				✓
PrinterMessageFromOperator	TextSyntax				✓
PrinterMoreInfo	URISyntax				✓
PrinterMoreInfoManufacturer	URISyntax				✓
PrinterName	TextSyntax				✓
PrinterResolution	ResolutionSyntax	✓	✓	✓	
PrinterState	PROCESSING, IDLE, STOPPED, UNKNOWN				✓
PrinterStateReason	COVER_OPEN, FUSER_OVER_TEMP, MEDIA_JAM и проч.				

Окончание табл. 10.5

Атрибут	Суперкласс или набор констант перечислимого типа	DA	PJA	PRA	PSA
`PrinterStateReasons`	`HashMap`				
`PrinterURI`	`URISyntax`				✓
`PrintQuality`	`DRAFT`, `NORMAL`, `HIGH`	✓	✓	✓	
`QueuedJobCount`	`IntegerSyntax`				✓
`ReferenceUriSchemesSupported`	`FILE`, `FTP`, `GOPHER`, `HTTP`, `HTTPS`, `NEWS`, `NNTP`, `WAIS`				
`RequestingUserName`	`TextSyntax`			✓	
`Severity`	`ERROR`, `REPORT`, `WARNING`				
`SheetCollate`	`COLLATED`, `UNCOLLATED`	✓	✓	✓	
`Sides`	`ONE_SIDED`, `DUPLEX` (или `TWO_SIDED_LONG_EDGE`), `TUMBLE` (или `TWO_SIDED_SHORT_EDGE`)	✓	✓	✓	

НА ЗАМЕТКУ! Как видите, многие из перечисленных в табл. 11.5 атрибутов печати являются узкоспециализированными и появились вследствие применения протокола Internet Printing Protocol 1.1 (по стандарту RFC 2911).

НА ЗАМЕТКУ! В более ранней версии прикладного интерфейса API для служб печати были внедрены классы **`JobAttributes`** и **`PageAttributes`**, которые служили той же цели, что и описываемые в этом разделе атрибуты печати. Но теперь эти классы считаются устаревшими.

`javax.print.attribute.Attribute` 1.4

- **`Class getCategory()`**
  Получает категорию данного атрибута печати.
- **`String getName()`**
  Получает имя данного атрибута печати.

`javax.print.attribute.AttributeSet` 1.4

- **`boolean add(Attribute attr)`**
  Вводит атрибут печати в набор атрибутов. Если в наборе имеется другой атрибут печати из той же категории, он замещается новым атрибутом. Возвращает логическое значение **`true`**, если в результате выполнения данной операции набор атрибутов изменился.
- **`Attribute get(Class category)`**
  Извлекает атрибут печати с ключом данной категории или пустое значение **`null`**, если такого атрибута не существует.

javax.print.attribute.AttributeSet 1.4 *(окончание)*

- **boolean remove(Attribute *attr*)**
- **boolean remove(Class *category*)**

  Удаляют заданный атрибут печати или атрибут указанной категории из набора атрибутов. Возвращают логическое значение **true**, если в результате выполнения данной операции набор атрибутов изменился.
- **Attribute[] toArray()**

  Возвращает массив со всеми атрибутами печати из данного набора.

javax.print.PrintService 1.4

- **PrintServiceAttributeSet getAttributes()**

  Получает атрибуты данной службы печати.

javax.print.DocPrintJob 1.4

- **PrintJobAttributeSet getAttributes()**

  Получает атрибуты текущего задания на печать.

Вот и подошла к концу эта длинная глава, посвященная расширенным средствам AWT. В завершающей этот том главе будет рассмотрен совсем иной аспект программирования на Java: взаимодействие на одной и той же машине с платформенно-ориентированным кодом, написанным на другом языке программирования.

ГЛАВА 12

Платформенно-ориентированные методы

В этой главе...

- Вызов функции на C из программы на Java
- Числовые параметры и возвращаемые значения
- Строковые параметры
- Доступ к полям
- Кодирование сигнатур
- Вызов методов на Java
- Доступ к элементам массивов
- Обработка ошибок
- Применение прикладного интерфейса API для вызовов
- Практический пример обращения к реестру Windows

Несмотря на все выгоды, которые дает решение пользоваться только Java, возможны ситуации, когда необходимо создавать (или использовать) код, написанный на каком-то другом языке. Такой код обычно называется *платформенно-ориентированным*.

В частности, когда язык Java только появился, многие считали, что для ускорения работы критических частей прикладных программ на Java было бы выгоднее

использовать язык C или C++. Но на практике такой прием редко оказывался эффективным. Презентация Java, проведенная на конференции JavaOne в 1996 году, очень ясно это показала. Разработчики криптографической библиотеки из компании Sun Microsystems продемонстрировали, что скорость выполнения криптографических функций, реализованных только в коде Java, оказалась вполне удовлетворительной. Безусловно, она уступала скорости выполнения тех же самых функций, написанных на C, но отличие было не очень значительным. Дело в том, что реализация платформы Java оказалась намного более быстродействующей, чем сетевой ввод-вывод. Именно это и послужило настоящей причиной более низкой производительности.

Безусловно, у перехода на платформенно-ориентированный код имеются свои недостатки. Если часть прикладной программы написана на каком-нибудь другом языке, придется предоставить отдельную специализированную библиотеку для каждой платформы, которую планируется поддерживать. Код, который пишется на C или C++, не предусматривает защиты против некорректного обращения к оперативной памяти вследствие неправильно заданного указателя. Это означает, что можно очень легко написать такие платформенно-ориентированные методы, которые будут нарушать работу программы или заражать операционную систему зловредным кодом.

Таким образом, платформенно-ориентированный код рекомендуется использовать только в тех случаях, когда это действительно необходимо. В частности, существуют три случая, когда выбор платформенно-ориентированного кода может оказаться правильным.

- Прикладной программе требуется доступ к таким системным функциям или устройствам, которые недоступны на платформе Java.
- Имеется немалое количество проверенного или отлаженного кода на другом языке, а также известны способы его переноса на все требующиеся целевые платформы.
- В результате сопоставительных испытаний обнаружено, что код, написанный на Java, выполняется намного медленнее, чем эквивалентный ему код на другом языке.

В состав платформы Java входит специальный прикладной интерфейс API для организации взаимодействия с платформенно-ориентированным кодом на C. Этот прикладной интерфейс называется JNI (Java Native Interface — платформенно-ориентированный интерфейс Java). И в этой главе обсуждаются вопросы программирования в прикладном интерфейсе JNI.

НА ЗАМЕТКУ C++! Для написания платформенно-ориентированных методов вместо C можно также пользоваться языком C++. Он имеет ряд преимуществ, обеспечивая, например, более строгий контроль соответствия типов и более удобный доступ к функциям JNI. Но в прикладном интерфейсе JNI не поддерживается взаимное преобразование классов Java и C++.

12.1. Вызов функции на C из программы на Java

Допустим, имеется некоторая функция, написанная на C и выполняющая что-нибудь полезное, но по той или иной причине ее нежелательно реализовывать

заново на Java. Ради простоты предположим для начала, что это элементарная функция, написанная на C, для вывода приветственного сообщения.

В языке программирования Java для организации вызова платформенно-ориентированного метода служит ключевое слово `native`, а сам метод, очевидно, вводится в класс. В листинге 12.1 показано, как именно это делается. Ключевое слово `native` предупреждает компилятор, что метод будет определяться внешним образом. Разумеется, платформенно-ориентированные методы не должны содержать код, написанный на Java, поэтому объявление метода состоит только из одного заголовка, после которого сразу же следует завершающая точка с запятой. Благодаря этому объявления платформенно-ориентированных методов становятся похожими на объявления абстрактных методов.

Листинг 12.1. Исходный код из файла **helloNative/HelloNative.java**

```
1 /**
2  * @version 1.11 2007-10-26
3  * @author Cay Horstmann
4  */
5 class HelloNative
6 {
7   public static native void greeting();
8 }
```

НА ЗАМЕТКУ! Ради простоты в примерах программ, приведенных в этой главе, пакеты не применяются.

В данном конкретном примере платформенно-ориентированный метод объявляется и как статический (`static`). Платформенно-ориентированные методы могут быть как статическими, так и нестатическими. Их рассмотрение было начато со статического метода, чтобы не усложнять дело передачей параметров.

Класс из приведенного выше примера можно скомпилировать, но если употребить его в программе, то виртуальная машина Java уведомит, что ей неизвестно, как найти функцию `greeting()`, выдав исключение типа `UnsatisfiedLinkError`. Чтобы устранить этот недостаток, следует реализовать платформенно-ориентированный код, т.е. написать соответствующую функцию на C. Имя этой функции должно *точно* соответствовать ожиданиям виртуальной машины Java. Для этого необходимо соблюсти следующие правила.

1. Использовать полное имя метода из кода Java, т.е. `HelloNative.greeting`. Если класс находится в пакете, добавить перед именем метода имя этого пакета, например `com.horstmann.HelloNative.greeting`.
2. Заменить все точки знаками подчеркивания и присоединить префикс `Java_`, например `Java_HelloNative_greeting` или `Java_com_horstmann_HelloNative_greeting`.
3. Если в имени класса присутствуют символы, не являющиеся ни буквами, ни цифрами в коде ASCII (например, знак `_` или `$`), или же символы в Юникоде свыше `'\u007F'`, заменить их последовательностью символов `_0xxxx`,

указав вместо *xxxx* четырехзначное шестнадцатеричное значение заменяемого символа в Юникоде.

НА ЗАМЕТКУ! Если платформенно-ориентированные методы *перегружаются*, т.е. предоставляется несколько платформенно-ориентированных методов с одинаковым именем, то в конце имени каждого из них следует присоединить два знака подчеркивания и закодированные типы аргументов. (Подробнее о кодировании типов аргументов речь пойдет далее в этой главе.) Так, если предоставляется один платформенно-ориентированный метод **greeting()** и другой платформенно-ориентированный метод **greeting(int *repeat*)**, первый из них должен именоваться как **Java_HelloNative_greeting__**, а другой — как **Java_HelloNative_greeting__I**.

Разумеется, вручную этого никто не делает. Для этого следует выполнить команду `javac` с параметром `-h`, указав каталог, в котором должны размещаться заголовочные файлы. Так, по приведенной ниже команде в текущем каталоге создается заголовочный файл `HelloNative.h`, содержимое которого представлено в листинге 12.2.

```
javac -h . HelloNative.java
```

Листинг 12.2. Исходный код из файла **helloNative/HelloNative.h**

```
1  /* ЭТОТ ФАЙЛ НЕ РЕДАКТИРУЕТСЯ - он
2     формируется автоматически */
3  #include <jni.h>
4  /* Заголовочный файл для класса HelloNative */
5
6  #ifndef _Included_HelloNative
7  #define _Included_HelloNative
8  #ifdef __cplusplus
9  extern "C" {
10 #endif
11 /*
12  * Класс: HelloNative
13  * Метод: greeting
14  * Сигнатура: ()V
15 */
16 JNIEXPORT void JNICALL Java_HelloNative_greeting
17   (JNIEnv *, jclass);
18
19 #ifdef __cplusplus
20 }
21 #endif
22 #endif
```

Как следует из листинга 12.2, данный файл содержит объявление функции `Java_HelloNative_greeting`. (Макрокоманды `JNIEXPORT` и `JNICALL` определяются в заголовочном файле `jni.h`. Они обозначают зависящие от компилятора спецификаторы для экспортируемых функций из динамически загружаемой библиотеки.)

Остается лишь скопировать прототип функции из заголовочного файла в файл исходного кода и написать для этой функции код реализации, как

показано в листинге 12.3. Не обращайте пока что внимания на аргументы `env` и `cl` в этой простой функции. Подробнее об аргументах платформенно-ориентированных функций и методов речь пойдет далее в этой главе.

Листинг 12.3. Исходный код из файла **helloNative/HelloNative.c**

```
1  /*
2   *@version 1.10 1997-07-01
3   * @author Cay Horstmann
4   */
5
6   #include "HelloNative.h"
7   #include <stdio.h>
8
9   JNIEXPORT void JNICALL Java_HelloNative_greeting(
10                         JNIEnv* env, jclass cl)
11 {
12   printf("Hello Native World!\n");
13 }
```

НА ЗАМЕТКУ C++! Для реализации платформенно-ориентированных методов можно использовать и язык C++. Но в этом случае функции, реализующие подобные методы, следует объявлять как **extern "C"**, как выделено полужирным в приведенном ниже примере кода.

```
extern "C"
JNIEXPORT void JNICALL Java_HelloNative_greeting(
                         JNIEnv* env, jclass cl)
{
  cout << "Hello, Native World!" << endl;
}
```

Это не позволит компилятору C++ "скорректировать" имена методов.

Далее платформенно-ориентированный код на C следует скомпилировать в динамически загружаемую библиотеку. Этот процесс зависит от используемого компилятора. Например, для вызова компилятора GNU C в ОС Linux соответствующая команда будет выглядеть следующим образом:

```
gcc -fPIC -I jdk/include -I jdk/include/linux -shared \
    -o libHelloNative.so HelloNative.c
```

Для вызова компилятора C от корпорации Microsoft в ОС Windows эта же команда примет следующий вид:

```
cl -I jdk\include -I jdk\include\win32 -LD HelloNative.c \
   -FeHelloNative.dll
```

где *jdk* обозначает каталог, в котором находится комплект JDK.

СОВЕТ. Если используется компилятор C от корпорации Microsoft, необходимо запустить сначала командный файл **vcvars32.bat** или **vsvars32.bat** из командной оболочки. Этот командный файл автоматически установит путь и переменные окружения, требующиеся для компилятора. Найти этот файл можно в каталоге **c:\Program Files\Microsoft Visual Studio 14.0\Common7\tools** или в каком-нибудь другом с аналогичным названием. Подробнее об этом см. в документации на применяемую версию Visual Studio.

Кроме того, можно воспользоваться бесплатно распространяемой средой программирования Cygwin, доступной для загрузки по адресу `http://www.cygwin.com`. В ее состав входит компилятор GNU C, а также библиотеки для написания кода в стиле Unix под ОС Windows. Если используется среда Cygwin, то команда вызова компилятора C будет выглядеть следующим образом:

```
gcc -mno-cygwin -D __int64="long long" -I jdk/include/ \
    -I jdk/include/win32 -shared -Wl,--add-stdcall-alias \
    -o HelloNative.dll HelloNative.c
```

НА ЗАМЕТКУ! В версии заголовочного файла **jni_md.h** для Windows содержится следующее объявление типа, характерное для компилятора C от корпорации Microsoft:

```
typedef __int64 jlong;
```

Следовательно, если используется компилятор GNU C, может возникнуть потребность отредактировать данный файл, например, следующим образом:

```
#ifdef __GNUC__
  typedef long long jlong;
#else
  typedef __int64 jlong;
#endif
```

С другой стороны, можно указать параметр командной строки **-D __int64="long long"** при вызове компилятора.

Наконец, следует ввести в прикладную программу вызов метода `System.loadLibrary()`. Для полной гарантии, что виртуальная машина будет загружать библиотеку перед первым использованием класса, следует организовать статический блок инициализации, как показано в листинге 12.4.

Листинг 12.4. Исходный код из файла **helloNative/HelloNativeTest.java**

```
1  /**
2   * @version 1.11 2007-10-26
3   * @author Cay Horstmann
4   */
5  class HelloNativeTest
6  {
7  public static void main(String[] args)
8    {
9      HelloNative.greeting();
10   }
11
12   static
13   {
14     System.loadLibrary("HelloNative");
15   }
16 }
```

На рис. 12.1 приведена общая схема обработки платформенно-ориентированного кода. После компиляции и запуска рассматриваемой здесь программы на экране появится сообщение `"Hello, Native World!"` (Здравствуй, родной мир).

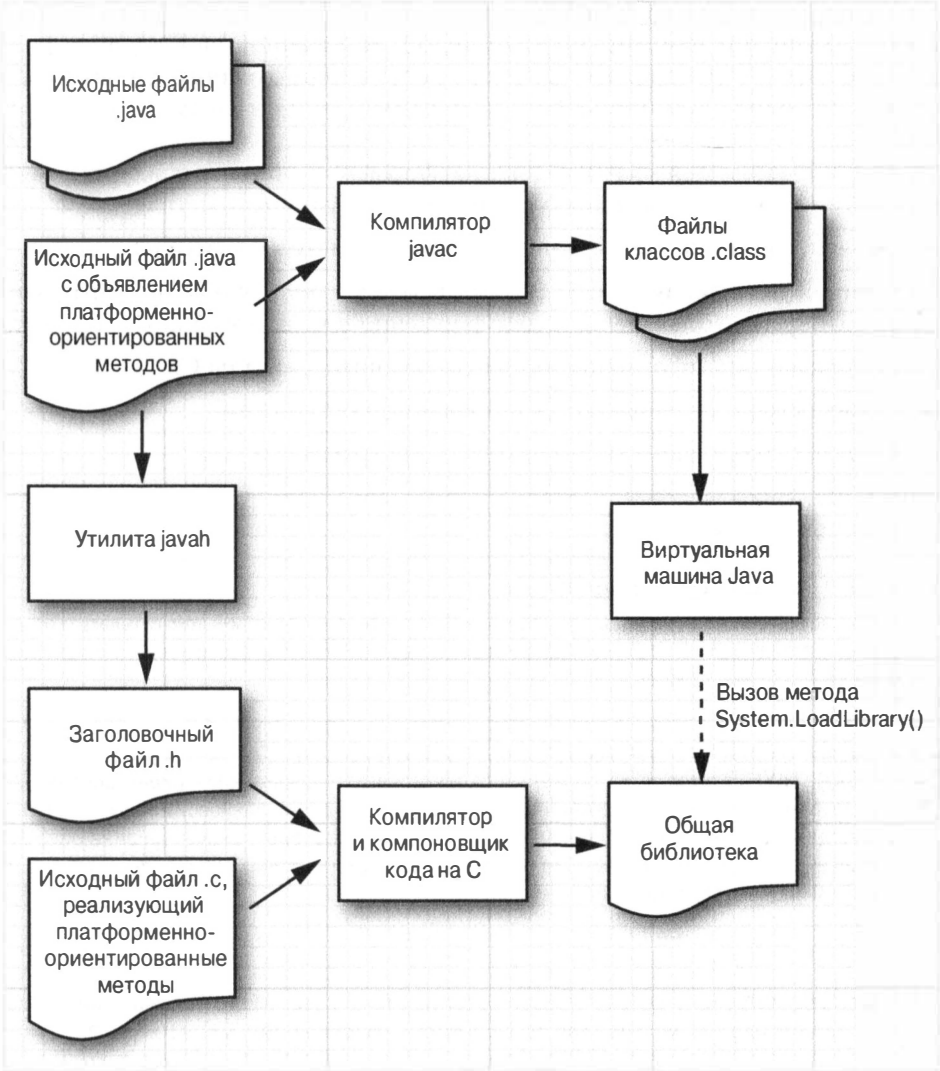


Рис. 12.1. Обработка платформенно-ориентированного кода

НА ЗАМЕТКУ! Пользователям ОС Linux придется включить текущий каталог в путь к библиотеке. Для этого им придется установить переменную окружения **LD_LIBRARY_PATH** следующим образом:

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

или же установить системное свойство **java.library.path**, как показано ниже.

```
java -Djava.library.path=. HelloNativeTest
```

Безусловно, результат не особенно впечатляет. Но если вспомнить, что выводимое сообщение формируется командой, написанной на C, а не на Java, то сразу становится очевидно, что это лишь первые шаги, которые удалось сделать на пути к преодолению пропасти между этими двумя языками программирования!

Таким образом, для связывания платформенно-ориентированного метода с программой на Java необходимо выполнить следующие действия.

1. Объявить платформенно-ориентированный метод в классе Java.
2. Выполнить команду `javac . -h` *имя_файла*`.java`, чтобы получить заголовочный файл с объявлением платформенно-ориентированного метода на C.
3. Реализовать платформенно-ориентированный метод на C.
4. Разместить полученный код в разделяемой библиотеке.
5. Загрузить разделяемую библиотеку в программу на Java.

java.lang.System 1.0

- **void loadLibrary(String *libname*)**

  Загружает библиотеку по указанному имени. Библиотека находится в том каталоге, который указан в пути поиска библиотек. Конкретный способ поиска библиотеки зависит от используемой операционной системы.

НА ЗАМЕТКУ! Некоторые разделяемые библиотеки с платформенно-ориентированным кодом требуют выполнения кода инициализации. Любой код инициализации можно разместить в методе **JNI_OnLoad()**. По завершении своей работы виртуальная машина Java будет аналогичным образом вызывать метод **JNI_OnUnload()**, если, конечно, предоставить его. Ниже приведены прототипы этих методов.

```
jint JNI_OnLoad(JavaVM* vm, void* reserved);
void JNI_OnUnload(JavaVM* vm, void* reserved);
```

Метод **JNI_OnLoad()** должен возвратить самую раннюю версию виртуальной машины, которая требуется для нормальной работы библиотеки, например **JNI_VERSION_1_2**.

12.2. Числовые параметры и возвращаемые значения

Для обмена числовыми параметрами между кодами на C и Java следует ясно понимать, какие типы параметров соответствуют друг другу. Например, в C имеются типы данных `int` и `long`, но их реализация зависит от конкретной платформы. На одних платформах тип данных `int` представляет 16-разрядное целочисленное значение, а на других — 32-разрядное. В Java тип данных `int` всегда представляет только 32-разрядное целочисленное значение. Именно поэтому в прикладном интерфейсе JNI и предоставляются такие типы данных, как `jint`, `jlong` и т.д. В табл. 12.1 приведено соответствие типов данных в Java и C.

Таблица 12.1. Типы данных в Java и C

Java	C	Количество байтов
`boolean`	`jboolean`	1
`byte`	`jbyte`	1
`char`	`jchar`	2
`short`	`jshort`	2
`int`	`jint`	4
`long`	`jlong`	8
`float`	`jfloat`	4
`double`	`jdouble`	8

В заголовочном файле `jni.h` эти типы данных объявляются с помощью операторов `typedef` в качестве эквивалентных типов целевой платформы. В этом же файле определяются также константы `JNI_FALSE = 0` и `JNI_TRUE = 1`.

До версии Java 5.0 в Java не было непосредственного аналога функции `printf()` на C. В приводимых ниже примерах предполагается, что по той или иной причине используется только старая версия JDK, и поэтому решено реализовать аналогичные функциональные возможности благодаря вызову функции `printf()` на C из платформенно-ориентированного метода. В листинге 12.5 представлен исходный код класса `Printf1`, использующего платформенно-ориентированный метод для вывода числового значения с плавающей точкой, указанной точностью и шириной поля.

Листинг 12.5. Исходный код из файла **`printf1/Printf1.java`**

```
1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4   */
5  class Printf1
6  {
7    public static native int print(int width, int precision, double x);
8
9  static
10  {
11    System.loadLibrary("Printf1");
12   }
13 }
```

Следует, однако, иметь в виду, что при реализации платформенно-ориентированного метода на C все параметры типа `int` и `double` заменяются параметрами типа `jint` и `jdouble`, как демонстрируется в листинге 12.6.

Листинг 12.6. Исходный код из файла **`printf1/Printf1.c`**

```
1  /**
2   @version 1.10 1997-07-01
3   @author Cay Horstmann
4  */
```

```
5
6  #include "Printf1.h"
7  #include <stdio.h>
8
9  JNIEXPORT jint JNICALL Java_Printf1_print(JNIEnv* env,
10                         jclass cl, jint width,
11                         jint precision, jdouble x)
12 {
13   char fmt[30];
14   jint ret;
15   sprintf(fmt, "%%%d.%df", width, precision);
16   ret = printf(fmt, x);
17   fflush(stdout);
18   return ret;
19 }
```

В функции из листинга 12.6 сначала компонуется форматирующая строка "%w.*pf*" с помощью переменной `fmt`, а затем вызывается стандартная функция `printf()`, после чего возвращается количество выведенных символов. В листинге 12.7 приведен исходный код примера программы, где демонстрируется применение класса `Printf1`.

Листинг 12.7. Исходный код из файла **`printf1/Printf1Test.java`**

```
1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4   */
5  class Printf1Test
6  {
7    public static void main(String[] args)
8    {
9      int count = Printf1.print(8, 4, 3.14);
10     count += Printf1.print(8, 4, count);
11     System.out.println();
12     for (int i = 0; i < count; i++)
13       System.out.print("-");
14     System.out.println();
15   }
16 }
```

12.3. Строковые параметры

Теперь рассмотрим, как обмениваться символьными строками с платформенно-ориентированными методами. Как известно, в Java символьные строки представляют собой последовательности кодовых точек в кодировке UTF-16, а в C они оканчиваются нулевым символом последовательности байтов. Иными словами, способы представления символьных строк в этих двух языках программирования заметно отличаются. Поэтому в прикладном интерфейсе JNI предоставляются два ряда функций для манипулирования символьными строками: первый из них позволяет преобразовывать символьные строки Java в байтовые

последовательности "модифицированного формата UTF-8", а второй — в массивы символьных значений в кодировке UTF-16, т.е. в массивы типа `jchar`. (Подробнее о кодировках UTF-8 и UTF-16, а также о "модифицированном формате UTF-8" см. в главе 2. Напомним, что в кодировке UTF-8 и "модифицированном формате UTF-8" символы кода ASCII оставляются без изменений, а все остальные символы Юникода кодируются в виде многобайтовых последовательностей.)

НА ЗАМЕТКУ! Стандартный и модифицированный форматы UTF-8 отличаются только дополнительными символами с кодами свыше **0xFFFF**. В стандартной кодировке UTF-8 эти символы кодируются 4-байтовыми последовательностями. В модифицированном формате символ сначала кодируется в виде так называемой "суррогатной пары" в кодировке UTF-16, а затем каждый суррогат — в кодировке UTF-8. В итоге получается последовательность из 6 байт. Такое решение нельзя назвать изящным, но другого выхода нет, ведь спецификация виртуальной машины Java была написана в те времена, когда длина символа в Юникоде ограничивалась 16 битами.

Если в прикладном коде на C уже применяются символы в Юникоде, то лучше всего воспользоваться вторым рядом функций преобразования символьных строк. Если же все строки содержат только символы в коде ASCII, то можно воспользоваться функциями преобразования символьных строк в "модифицированный формат UTF-8".

Платформенно-ориентированный метод со строковым параметром типа `String` фактически получает значение типа `jstring`. Платформенно-ориентированный метод с возвращаемым строковым значением типа `String` должен возвращать значение типа `jstring`. Для чтения и создания объектов типа `jstring` применяются специальные функции JNI. Например, функция `NewStringUTF()` создает новый объект типа `jstring` из массива `char`, содержащего символы в коде ASCII или, что бывает намного чаще, байтовые последовательности, кодированные в "модифицированном формате UTF-8". Функции JNI вызываются не совсем обычно. Например, вызов функции `NewStringUTF()` выглядит следующим образом:

```
JNIEXPORT jstring JNICALL Java_HelloNative_getGreeting(
                          JNIEnv* env, jclass cl)
{
  jstring jstr;
  char greeting[] = "Hello, Native World\n";
  jstr = (*env)->NewStringUTF(env, greeting);
  return jstr;
}
```

НА ЗАМЕТКУ! Весь код здесь и далее в этой главе приводится на C, если не указано иное.

Во всех вызовах функций JNI используется указатель `env`, который является первым параметром каждого платформенно-ориентированного метода. Он ссылается на таблицу с указателями функций (рис. 12.2). Вследствие этого каждый вызов функции JNI следует предварять префиксом `(*env)->` для разыменования

конкретного указателя функции. Более того, указатель `env` является также первым параметром каждой функции JNI.

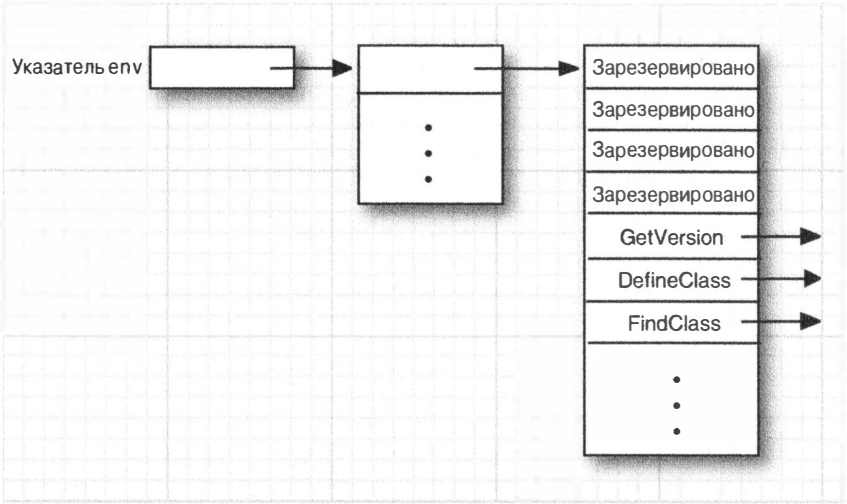


Рис. 12.2. Указатель **env**

НА ЗАМЕТКУ C++! В языке C++ доступ к функциям JNI организуется проще. В версии класса **`JNIEnv`** на C++ имеются встраиваемые функции-члены, способные автоматически выполнять поиск указателей функций. Например, вызвать функцию **`NewStringUTF()`** в C++ можно следующим образом:

```
jstr = env->NewStringUTF(greeting);
```

Обратите внимание на отсутствие указателя **`JNIEnv`** в списке параметров вызываемого метода.

Функция `NewStringUTF()` позволяет создавать новый объект типа `jstring`. Для считывания содержимого уже существующего объекта типа `jstring` применяется функция `GetStringUTFChars()`, которая возвращает указатель `const jbyte*` на описывающие строку символы в модифицированном формате UTF-8. Однако некоторые виртуальные машины Java могут использовать данный формат для внутреннего представления символьных строк, из-за чего возможно получение указателя на конкретную символьную строку в Java. А поскольку символьные строки в Java считаются неизменяемыми, то очень важно со *всей* серьезностью отнестись к обозначению `const` и не пытаться ничего записывать в такой символьный массив. С другой стороны, если в виртуальной машине применяется кодировка UTF-16 или UTF-32 для внутреннего представления строк, то вызов функции `GetStringUTFChars()` приведет к выделению нового блока памяти и его заполнению эквивалентными символами в модифицированном формате UTF-8.

По завершении манипулирования символьной строкой следует вызвать функцию `ReleaseStringUTFChars()`, чтобы сообщить об этом виртуальной машине, которая должна освободить память, занимаемую данной строкой. Как известно, процесс "сборки мусора" происходит в отдельном потоке исполнения и может

прерывать выполнение платформенно-ориентированных методов. Именно поэтому требуется вызвать функцию `ReleaseStringUTFChars()`.

С другой стороны, можно предоставить свой буфер для хранения символов строки, вызвав функции `GetStringRegion()` или `GetStringUTFRegion()`. Наконец, функция `GetStringUTFLength()` возвращает количество символов, необходимых для кодирования символьной строки в модифицированном формате UTF-8.

НА ЗАМЕТКУ! Описание прикладного интерфейса JNI API можно найти по адресу **https://docs.oracle.com/javase/7/docs/technotes/guides/jni**.

Доступ к символьным строкам в Java из кода на C

- **jstring NewStringUTF(JNIEnv* *env*, const char *bytes*[])**

  Возвращает новый объект символьной строки в Java из оканчивающейся нулевым байтом последовательности байтов в модифицированном формате UTF-8, или значение **NULL**, если не удается сконструировать символьную строку.

- **jsize GetStringUTFLength(JNIEnv* *env*, jstring *string*)**

  Возвращает количество байтов, требующееся для кодирования символьной строки в модифицированном формате UTF-8 (без учета завершающего нулевого байта).

- **const jbyte* GetStringUTFChars(JNIEnv* *env*, jstring *string*, jboolean* *isCopy*)**

  Возвращает указатель на символьную строку в модифицированном формате UTF-8 или значение **NULL**, если создать символьный массив в таком формате не удается. Этот указатель остается действительным до тех пор, пока не будет вызвана функция **ReleaseStringUTFChars()**. Параметр ***isCopy*** принимает значение **NULL** или указатель на переменную типа **jboolean**, которая заполняется значением **JNI_TRUE**, если создается копия символьной строки, а иначе — значение **JNI_FALSE**.

- **void ReleaseStringUTFChars(JNIEnv* *env*, jstring *string*, const jbyte *bytes*[])**

  Извещает виртуальную машину, что из платформенно-ориентированного кода больше не требуется доступ к символьной строке Java через массив ***bytes*** по указателю, возвращаемому функцией **GetStringUTFChars()**.

- **void GetStringRegion(JNIEnv* *env*, jstring *string*, jsize *start*, jsize *length*, jchar* *buffer*)**

  Копирует последовательность двойных байтов в кодировке UTF-16 из символьной строки в предоставляемый пользователем буфер размером, как минимум, **2×*length***.

- **void GetStringUTFRegion(JNIEnv* *env*, jstring *string*, jsize *start*, jsize *length*, jbyte* *buffer*)**

  Копирует последовательность байтов в модифицированном формате UTF-8 из символьной строки в буфер, предоставляемый пользователем. Емкости буфера должно хватить, чтобы вместить все байты. В худшем случае копируется только **3×*length*** байтов.

- **jstring NewString(JNIEnv* *env*, const jchar *chars*[], jsize *length*)**

  Возвращает новый объект символьной строки Java из символьной строки в Юникоде или значение **NULL**, если создать символьную строку Java не удается.

Доступ к символьным строкам в Java из кода на C *(окончание)*

- **`jsize GetStringLength(JNIEnv* env, jstring string)`**

  Возвращает количество символов в строке.

- **`const jchar* GetStringChars(JNIEnv* env, jstring string, jboolean* isCopy)`**

  Возвращает указатель на последовательность символов в Юникоде или значение **`NULL`**, если создать символьный массив в таком формате не удается. Указатель действует до тех пор, пока не будет вызвана функция **`ReleaseStringUTFChars()`**. Параметр ***`isCopy`*** принимает значение **`NULL`** или указатель на переменную типа **`jboolean`**, которая заполняется значением **`JNI_TRUE`**, если создается копия символьной строки, а иначе — значением **`JNI_FALSE`**.

- **`void ReleaseStringChars(JNIEnv* env, jstring string, const jchar chars[])`**

  Извещает виртуальную машину, что из платформенно-ориентированного кода больше не требуется доступ к символьной строке Java через массив ***`chars`*** по указателю, возвращаемому функцией **`GetStringChars()`**.

Теперь опробуем упомянутые выше функции на практике, разработав класс, способный вызывать функцию `sprintf()` на C. Желательно, чтобы эта функция вызывалась так, как показано в листинге 12.8.

Листинг 12.8. Исходный код из файла **`printf2/Printf2Test.java`**

```
1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4   */
5  class Printf2Test
6  {
7    public static void main(String[] args)
8    {
9      double price = 44.95;
10     double tax = 7.75;
11     double amountDue = price * (1 + tax / 100);
12
13     String s = Printf2.sprint("Amount due = %8.2f", amountDue);
14     System.out.println(s);
15   }
16 }
```

В листинге 12.9 представлен исходный код класса с платформенно-ориентированным методом `sprint()`.

Листинг 12.9. Исходный код из файла **`printf2/Printf2.java`**

```
1  /**
2   * @version 1.10 1997-07-01
3   * @author Cay Horstmann
4   */
```

```
5  class Printf2
6  {
7    public static native String sprint(String format, double x);
8
9   static
10   {
11     System.loadLibrary("Printf2");
12   }
13 }
```

Таким образом, функция на C, с помощью которой форматируется число с плавающей точкой, имеет следующий прототип:

```
JNIEXPORT jstring JNICALL Java_Printf2_sprint(JNIEnv* env,
                  jclass cl, jstring format, jdouble x)
```

Код, реализующий эту функцию на C, приведен в листинге 12.10. Следует, однако, иметь в виду, что для считывания параметра форматирования `format` вызывается функция `GetStringUTFChars()`, для генерирования возвращаемого значения — функция `NewStringUTF()`, а для извещения виртуальной машины о том, что доступ к символьной строке больше не требуется, — функция `ReleaseStringUTFChars()`.

Листинг 12.10. Исходный код из файла **`printf2/Printf2.c`**

```
1  /**
2   @version 1.10 1997-07-01
3   @author Cay Horstmann
4  */
5
6  #include "Printf2.h"
7  #include <string.h>
8  #include <stdlib.h>
9  #include <float.h>
10
11 /**
12   * @param format Символьная строка со спецификатором
13   * формата для функции printf(), например, "%8.2f".
14   * Подстроки "%%" пропускаются
15   * @return Возвращает указатель на спецификатор формата,
16   *         пропуская символ '%', или значение NULL в
17   *         отсутствие однозначного спецификатора формата
18   */
19 char* find_format(const char format[])
20 {
21   char* p;
22   char* q;
23
24   p = strchr(format, '%');
25   while (p != NULL && *(p + 1) == '%')
26          /* пропустить подстроку "%%" */
27     p = strchr(p + 2, '%');
28   if (p == NULL) return NULL;
```

```
29        /* проверить единственность символа "%" */
30    p++;
31    q = strchr(p, '%');
32    while (q != NULL && *(q + 1) == '%')
33          /* пропустить подстроку "%%" */
34      q = strchr(q + 2, '%');
35    if (q != NULL)
36      return NULL; /* символ "%" не единственный */
37    q = p + strspn(p, " -0+#"); /* пропустить признаки */
38    q += strspn(q, "0123456789");
39        /* пропустить ширину поля */
40    if (*q == '.')
41      { q++; q += strspn(q, "0123456789"); }
42        /* пропустить точность */
43    if (strchr("eEfFgG", *q) == NULL)
44      return NULL; /* это не формат с плавающей точкой */
45    return p;
46  }
47
48  JNIEXPORT jstring JNICALL Java_Printf2_sprint(
49                    JNIEnv* env, jclass cl,
50                    jstring format, jdouble x)
51  {
52    const char* cformat;
53    char* fmt;
54    jstring ret;
55
56    cformat = (*env)->
57              GetStringUTFChars(env, format, NULL);
58    fmt = find_format(cformat);
59    if (fmt == NULL)
60      ret = format;
61    else
62    {
63      char* cret;
64      int width = atoi(fmt);
65      if (width == 0) width = DBL_DIG + 10;
66      cret = (char*) malloc(strlen(cformat) + width);
67      sprintf(cret, cformat, x);
68      ret = (*env)->NewStringUTF(env, cret);
69      free(cret);
70    }
71    (*env)->ReleaseStringUTFChars(env, format, cformat);
72    return ret;
73  }
```

В данной функции сохранен простой механизм обработки ошибок. Если код форматирования, отвечающий за вывод числового значения с плавающей точкой, не соответствует форме `%w.pc` (где `c` — один из символов `e`, `E`, `f`, `g` или `G`), тогда форматирование числового значения не выполняется. Далее в этой главе будет показано, как организовать в платформенно-ориентированном методе генерирование исключения.

12.4. Доступ к полям

Все рассматривавшиеся до сих пор платформенно-ориентированные методы были статическими с числовыми и строковыми параметрами. Теперь речь пойдет о методах, способных обращаться с объектами произвольного типа. Для примера попробуем реализовать как платформенно-ориентированный метод из класса `Employee`, упоминавшийся в главе 4 первого тома настоящего издания. И хотя это несколько отвлеченный пример, он наглядно показывает, как получается доступ к полям из платформенно-ориентированного метода.

12.4.1. Доступ к полям экземпляра

Чтобы выяснить, как получить доступ к полям экземпляра из платформенно-ориентированного метода, реализуем снова метод `raiseSalary()`. Исходный код этого метода на Java выглядит следующим образом:

```
public void raiseSalary(double byPercent)
{
  salary *= 1 + byPercent / 100;
}
```

Перепишем этот метод как платформенно-ориентированный. В отличие от платформенно-ориентированных методов из предыдущих примеров, данный метод не является статическим. Поэтому в результате выполнения утилиты `javah` получается следующий прототип данного метода:

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(
                        JNIEnv *, jobject, jdouble);
```

Обратите внимание на второй параметр, который теперь имеет тип `jobject`, а не `jclass`. По существу, он является эквивалентом ссылки `this`. Статические методы получают ссылку на класс, а нестатические — на конкретный объект.

Итак, нам требуется доступ к полю `salary` объекта, представленного неявным параметром. В первоначальном варианте связывания кода Java и C в версии Java 1.0 подобная задача решалась очень просто. Программисты могли получить прямой доступ к полям данных объекта. Но для прямого доступа всем виртуальным машинам придется раскрыть внутреннюю структуру данных. Поэтому для получения и установки значений в полях программистам приходится обращаться к специальным функциям JNI.

В рассматриваемом здесь примере поле `salary` относится к типу `double`, поэтому для доступа к нему воспользуемся функциями `GetDoubleField()` и `SetDoubleField()`. Для доступа к полям данных других типов предусмотрены функции `GetIntField()` и `SetIntField()`, `GetObjectField()` и `SetObjectField()` и т.д. При обращении с ними применяется следующий синтаксис:

```
x = (*env)->GetXxxField(env, this_obj, fieldID);
(*env)->SetXxxField(env, this_obj, fieldID, x);
```

где параметр `fieldID` представляет значение специального типа, которое обозначает поле структуры, а суффикс *Xxx* — тип данных Java (например, `Object`, `Boolean`, `Byte` и т.д.). Чтобы получить значение `fieldID`, нужно сначала получить

значение, представляющее класс, используя функцию GetObjectClass() или FindClass(). В частности, функция GetObjectClass() возвращает класс произвольного объекта, как показано ниже.

```
jclass class_Employee =
       (*env)->GetObjectClass(env, this_obj);
```

Функция FindClass() позволяет указать имя класса в виде символьной строки (вместо точки для разделения пакетов следует использовать знак /), как выделено ниже полужирным.

```
jclass class_String =
       (*env)->FindClass(env, "java/lang/String");
```

Для получения параметра fieldID служит функция GetFieldID(), при вызове которой указывается *имя* поля и его *сигнатура*, т.е. кодированное представление его типа. Например, в приведенной ниже строке кода получается идентификатор поля salary.

```
jfieldID id_salary = (*env)->
         GetFieldID(env, class_Employee, "salary", "D");
```

Символьная строка "D" обозначает здесь тип double. Более подробно правила кодирования сигнатур полей рассматриваются в следующем разделе.

На первый взгляд, такой порядок доступа к полям данных может показаться слишком сложным. Не следует, однако, забывать, что разработчики прикладного интерфейса JNI не стремились раскрывать поля данных напрямую, поэтому им пришлось предоставить функции для установки и получения значений полей. Для сокращения издержек, связанных с применением этих функций, вычисление идентификатора из имени поля выделено в отдельную задачу, поскольку она требует наибольших издержек вычислительных ресурсов. Следовательно, при неоднократном обращении к полю с целью установить или получить его значение идентификатор этого поля вычисляется только один раз.

Итак, принимая во внимание все сказанное выше, реализуем метод raise Salary() как платформенно-ориентированный. Ниже приведен исходный код этого метода, написанный на C.

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary(
          JNIEnv* env, jobject this_obj, jdouble byPercent)
{
  /* получить класс */
  jclass class_Employee =
         (*env)->GetObjectClass(env, this_obj);
  /* получить идентификатор поля */
  jfieldID id_salary = (*env)->
        GetFieldID( env, class_Employee, "salary", "D");
  /* получить значение поля */
  jdouble salary = (*env)->
          GetDoubleField(env, this_obj, id_salary);
  salary *= 1 + byPercent / 100;
  /* установить значение поля */
  (*env)->SetDoubleField(env, this_obj, id_salary, salary);
}
```

ВНИМАНИЕ! Ссылки на классы действительны только до завершения платформенно-ориентированного метода. Поэтому кешировать значения, возвращаемые функцией **GetObjectClass()**, не удастся. Не пытайтесь сберечь ссылку на класс для последующего использования при вызове платформенно-ориентированного метода. Всякий раз, когда используется платформенно-ориентированный метод, следует вызывать функцию **GetObjectClass()**. Если же это неприемлемо, ссылку можно зафиксировать с помощью функции **NewGlobalRef()**, например, следующим образом:

```
static jclass class_X = 0;
static jfieldID id_a;
. . .
if (class_X == 0)
{
  jclass cx = (*env)->GetObjectClass(env, obj);
  class_X = (*env)->NewGlobalRef(env, cx);
  id_a = (*env)->GetFieldID(env, cls, "a", ". . .");
}
```

Теперь зафиксированную ссылку на класс и идентификаторы полей можно использовать при последующих вызовах платформенно-ориентированного метода. По завершении всех необходимых операций с классом, доступным по данной ссылке, ее следует удалить с помощью функции **DeleteGlobalRef()**:

```
(*env)->DeleteGlobalRef(env, class_X);
```

В листинге 12.11 приведен исходный код примера программы на Java, а в листинге 12.12 — исходный код класса `Employee`. Что же касается исходного кода платформенно-ориентированного метода `raiseSalary()` на C, то он представлен в листинге 12.13.

Листинг 12.11. Исходный код из файла **employee/EmployeeTest.java**

```
1  /**
2   * @version 1.11 2018-05-01
3   * @author Cay Horstmann
4   */
5
6  public class EmployeeTest
7  {
8    public static void main(String[] args)
9    {
10     var staff = new Employee[3];
11
12     staff[0] = new Employee("Harry Hacker", 35000);
13     staff[1] = new Employee("Carl Cracker", 75000);
14     staff[2] = new Employee("Tony Tester", 38000);
15
16     for (Employee e : staff)
17       e.raiseSalary(5);
18     for (Employee e : staff)
19       e.print();
20   }
21 }
```

Листинг 12.12. Исходный код из файла **employee/Employee.java**

```
1  /**
2   * @version 1.10 1999-11-13
3   * @author Cay Horstmann
4   */
5
6  public class Employee
7  {
8    private String name;
9    private double salary;
10
11   public native void raiseSalary(double byPercent);
12
13   public Employee(String n, double s)
14   {
15     name = n;
16     salary = s;
17   }
18
19   public void print()
20   {
21     System.out.println(name + " " + salary);
22   }
23
24   static
25   {
26     System.loadLibrary("Employee");
28   }
29 }
```

Листинг 12.13. Исходный код из файла **employee/Employee.c**

```
1  /**
2   * @version 1.10 1999-11-13
3   * @author Cay Horstmann
4   */
5
6  #include "Employee.h"
7
8  #include <stdio.h>
9
10 JNIEXPORT void JNICALL Java_Employee_raiseSalary(
11      JNIEnv* env, jobject this_obj, jdouble byPercent)
12 {
13   /* получить класс */
14   jclass class_Employee =
15         (*env)->GetObjectClass(env, this_obj);
16
17   /* получить идентификатор поля */
18   jfieldID id_salary = (*env)->
19         GetFieldID( env, class_Employee, "salary", "D");
20
21   /* получить значение поля */
```

```
22    jdouble salary = (*env)->
23            GetDoubleField(env, this_obj, id_salary);
24
25    salary *= 1 + byPercent / 100;
26
27    /* установить значение поля */
28    (*env)->SetDoubleField(
29            env, this_obj, id_salary, salary);
30 }
```

12.4.2. Доступ к статическим полям

Доступ к статическим полям осуществляется аналогично доступу к нестатическим полям. Для этой цели служат функции `GetStaticFieldID()` и `GetStaticXxxField()/SetStaticXxxField()`, которые действуют практически так же, как и их нестатические аналоги, но имеют два отличия.

- Вследствие того что объект отсутствует, для получения ссылки на класс следует вызвать функцию `FindClass()` вместо функции `GetObjectClass()`.
- Для доступа к статическому полю следует предоставить класс, а не объект.

В качестве примера ниже показано, как получить ссылку на стандартный поток вывода `System.out`.

```
/* получить класс */
jclass class_System =
       (*env)->FindClass(env, "java/lang/System");

/* получить идентификатор поля */
jfieldID id_out = (*env)->GetStaticFieldID(
                  env, class_System, "out",
                  "Ljava/io/PrintStream;");

/* получить значение поля */
jobject obj_out = (*env)->GetStaticObjectField(
                  env, class_System, id_out);
```

Доступ к полям

- **jfieldID GetFieldID(JNIEnv* *env*, jclass *cl*, const char *name*[], const char *fieldSignature*[])**

  Возвращает идентификатор поля в классе.
- ***Xxx* GetXxxField(JNIEnv* *env*, jobject *obj*, jfieldID *id*)**

  Возвращает значение поля. В качестве типа ***Xxx*** поля может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**.
- **void SetXxxField(JNIEnv* *env*, jobject *obj*, jfieldID *id*, *Xxx* value)**

  Устанавливает новое значение в поле. В качестве типа ***Xxx*** поля может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**.

Доступ к полям *(окончание)*

- **jfieldID GetStaticFieldID(JNIEnv* *env*, jclass *cl*, const char *name*[], const char *fieldSignature*[])**

  Возвращает идентификатор статического поля в классе.
- ***Xxx* GetStatic*Xxx*Field(JNIEnv* *env*, jclass *cl*, jfieldID *id*)**

  Возвращает значение статического поля. В качестве типа ***Xxx*** поля может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**.
- **void SetStaticXxxField(JNIEnv* *env*, jclass *cl*, jfieldID *id*, *Xxx* value)**

  Устанавливает новое значение в статическом поле. В качестве типа ***Xxx*** поля может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**.

12.5. Кодирование сигнатур

Для доступа к полям экземпляра и вызова методов, определенных в Java, необходимо знать правила корректирования имен типов данных и сигнатур методов. (Напомним, что сигнатура метода описывает параметры и возвращаемое значение.) Для этой цели используется приведенная ниже схема кодирования.

B	**byte**
C	**char**
D	**double**
F	**float**
I	**int**
J	**long**
L*имя_класса*;	тип класса
S	**short**
V	**void**
Z	**boolean**

Для описания типа массива служит знак `[`. Например, массив символьных строк описывается следующим образом:

```
[Ljava/lang/String;
```

Двумерный массив `float[][]` кодируется так:

```
[[F
```

Для полного обозначения сигнатуры метода сначала перечисляются все типы параметров в круглых скобках, а затем указывается возвращаемый тип. Например, сигнатура метода, получающего два целочисленных значения и возвращающего одно целочисленное значение, кодируется следующим образом:

```
(II)I
```

Метод `sprint()`, упоминавшийся в разделе 12.3, имеет приведенную ниже скорректированную сигнатуру. Она означает, что данный метод получает значение типа `String` и `double` и возвращает значение типа `String`.

```
(Ljava/lang/String;D)Ljava/lang/String;
```

Обратите внимание на то, что точка с запятой в конце выражения *Lимя_класса;* служит признаком окончания определяющего тип выражения, а не разделителем параметров. Например, следующий конструктор:

```
Employee(java.lang.String, double, java.util.Date)
```

имеет такую сигнатуру:

```
"(Ljava/lang/String;DLjava/util/Date;)V"
```

Обратите внимание на отсутствие разделителя между обозначениями `D` и `Ljava/util/Date;`, а также на то, что в данной схеме кодирования для разделения имен пакетов и классов требуется указывать знак косой черты (`/`), а не точки (`.`). Обозначение `V` в конце сигнатуры означает, что данный конструктор ничего не возвращает, поскольку возвращаемое значение имеет тип `void`. И хотя возвращаемый тип для конструкторов в Java не указывается, обозначение `V` все равно должно добавляться в конце сигнатуры специально для виртуальной машины.

СОВЕТ. Для автоматического генерирования сигнатур методов из файлов классов достаточно выполнить команду **javap** с параметром командной строки **-s**, например, следующим образом:

```
javap -s -private Employee
```

В итоге на экран будут выведены сигнатуры всех полей и методов указанного класса, как показано ниже.

```
Compiled from "Employee.java"
public class Employee extends java.lang.Object{
private java.lang.String name;
  Signature: Ljava/lang/String;
private double salary;
  Signature: D
public Employee(java.lang.String, double);
  Signature: (Ljava/lang/String;D)V
public native void raiseSalary(double);
  Signature: (D)V
public void print();
  Signature: ()V
static {};
  Signature: ()V
}
```

НА ЗАМЕТКУ! Не существует никакого рационального объяснения тому, что программисты вынуждены применять именно такую скорректированную схему кодирования сигнатур. Разработчики механизма вызова платформенно-ориентированного кода с таким же успехом могли бы написать функцию, позволяющую сначала считывать сигнатуры в стиле языка программирования Java, например, **`void(int, java.lang.String)`**, а затем кодировать их в любое удобное внутреннее представление. Но, с другой стороны, применение скорректированных сигнатур позволяет приобщиться к таинствам программирования на уровне, близком к виртуальной машине.

12.6. Вызов методов на Java

Итак, вы знаете, как вызывать функции, написанные на C, из прикладного кода, написанного на Java. Именно для этого и служат платформенно-ориентированные методы. А можно ли выполнить обратную операцию, вызвав методы на Java из функций на C, и зачем вообще может потребоваться такая операция? Оказывается, что нередко из платформенно-ориентированного метода требуется запрашивать какую-нибудь службу из переданного ему объекта. Поэтому далее будет показано, как добиться этого сначала для методов экземпляра, а затем для статических методов.

12.6.1. Методы экземпляра

В качестве примера вызова метода экземпляра на Java из платформенно-ориентированного кода усовершенствуем класс `Printf`, введя в него метод, действующий как функция `fprintf()` на C. Следовательно, этот метод должен быть в состоянии выводить отформатированную строку в произвольный поток в виде объекта типа `PrintWriter`. Ниже показано, каким образом этот метод определяется на Java.

```
class Printf3
{
  public native static void fprint(PrintWriter out,
                                   String s, double x);
  . . .
}
```

Сначала выводимая символьная строка составляется в объекте `str` типа `String`, как и в реализованном ранее методе `sprint()`. Затем из функции на C вызывается метод `print()` из класса `PrintWriter`. Любой метод на Java можно вызвать из функции на C в следующей форме:

```
(*env)->Call Xxx Method(env, неявный_параметр,
        идентификатор_метода, явные_параметры)
```

где *Xxx* обозначает возвращаемый тип, например `Void`, `Int`, `Obect` и т.д. Если для доступа к полю требуется его идентификатор, то для вызова метода следует указать его идентификатор. Для получения идентификатора вызываемого метода в прикладном интерфейсе JNI предусмотрена функция `GetMethodID()` с параметрами, которые задают класс, имя и сигнатуру метода.

В рассматриваемом здесь примере требуется получить идентификатор метода `print()` из класса `PrintWriter`. В классе `PrintWriter` имеются девять разных методов под одинаковым именем `print`. Поэтому для выбора конкретного варианта следует точно указать параметры и возвращаемое значение. Например, для вызова метода `void print(java.lang.String)` следует закодировать его скорректированную сигнатуру в символьной строке `"(Ljava/lang/String;)V"`, как пояснялось в предыдущем разделе. Ниже приведен весь код для вызова этого метода.

```
/* получить класс */
class_PrintWriter = (*env)->GetObjectClass(env, out);
```

```
/* получить идентификатор метода */
id_print = (*env)->GetMethodID(env, class_PrintWriter,
                   "print", "(Ljava/lang/String;)V");

/* вызвать метод */
(*env)->CallVoidMethod(env, out, id_print, str);
```

В листинге 12.14 приведен исходный код примера программы на Java, а в листинге 12.15 — исходный код класса `Printf3`. Исходный код платформенно-ориентированного метода `fprint()` на C представлен в листинге 12.16.

НА ЗАМЕТКУ! Числовые идентификаторы методов и полей выполняют те же функции, что и объекты типа **`Method`** и **`Field`** из прикладного интерфейса API для рефлексии в Java. Для их взаимного преобразования служат следующие функции:

```
jobject ToReflectedMethod(JNIEnv* env, jclass class,
                          jmethodID methodID);
    // возвращает объект типа Method
methodID FromReflectedMethod(JNIEnv* env, jobject method);
jobject ToReflectedField(JNIEnv* env, jclass class,
                         jfieldID fieldID);
    // возвращает объект типа Field
fieldID FromReflectedField(JNIEnv* env, jobject field);
```

12.6.2. Статические методы

Вызов статических методов из платформенно-ориентированного кода выполняется аналогично вызову методов экземпляра, за исключением следующих отличий.

- Применяются функции `GetStaticMethodID()` и `CallStatic`*`Xxx`*`Method()`.
- При вызове метода указывается объект класса, а не объект в виде неявного параметра.

В качестве примера рассмотрим следующий вызов статического метода `getProperty()` из платформенно-ориентированного метода:

```
System.getProperty("java.class.path")
```

Из этого метода возвращается символьная строка, содержащая текущий путь к классу. Сначала необходимо обнаружить используемый класс, а в отсутствие объектов класса `System` — вызвать функцию `FindClass()` вместо функции `GetObjectClass()`, как показано ниже.

```
jclass class_System = (*env)->FindClass(env, "java/lang/System");
```

Затем следует получить идентификатор статического метода `getProperty()`. Параметр и возвращаемое значение этого метода представлены символьной строкой, поэтому его сигнатура кодируется следующим образом:

```
"(Ljava/lang/String;)Ljava/lang/String;"
```

Далее следует получить идентификатор данного метода:

```
jmethodID id_getProperty = (*env)->GetStaticMethodID(
          env, class_System, "getProperty",
          "(Ljava/lang/String;)Ljava/lang/String;");
```

Наконец, остается сделать вызов данного метода, как показано ниже. Следует, однако, иметь в виду, что функции `CallStaticObjectMethod()` передается объект, представляющий класс.

```
jobject obj_ret = (*env)->CallStaticObjectMethod(
        env, class_System, id_getProperty,
        (*env)->NewStringUTF(env, "java.class.path"));
```

Значение, возвращаемое из данного метода, относится к типу `jobject`. Поэтому для выполнения операций над символьными строками его придется привести к типу `jstring` следующим образом:

```
jstring str_ret = (jstring) obj_ret;
```

НА ЗАМЕТКУ C++! В языке C типы **jstring** и **jclass**, а также типы массивов, которые рассматриваются далее, равнозначны типу **jobject**, поэтому выполнять приведение типов в упомянутой выше строке кода на C необязательно. Но в языке C++ эти типы являются указателями на так называемые "фиктивные" классы, имеющие правильную иерархию наследования. Например, присваивание значения типа **jstring** переменной типа **jobject** допускается без приведения типов. Но в то же время приведение типов обязательно для присваивания значения типа **jobject** переменной типа **jstring**.

12.6.3. Конструкторы

В платформенно-ориентированном методе можно создать новый объект Java, вызвав его конструктор с помощью функции `NewObject()` в следующей форме:

```
jobject obj_new = (*env)->NewObject(env, класс,
                        идентификатор_метода,
                        параметры_конструктора);
```

Чтобы получить идентификатор метода, следует вызвать функцию `GetMethodID()`, указав имя метода в символьной строке "<init>" и закодированную сигнатуру конструктора с возвращаемым типом `void`. В качестве примера ниже показано, каким образом в платформенно-ориентированном методе создается объект типа `FileOutputStream` для потока вывода в файл.

```
const char[] fileName = ". . .";
jstring str_fileName = (*env)->NewStringUTF(env, fileName);
jclass class_FileOutputStream =
     (*env)->FindClass(env, "java/io/FileOutputStream");
jmethodID id_FileOutputStream = (*env)->GetMethodID(
                env, class_FileOutputStream,
                "<init>", "(Ljava/lang/String;)V");
jobject obj_stream = (*env)->NewObject(
                    env, class_FileOutputStream,
                    id_FileOutputStream, str_fileName);
```

Обратите внимание на то, что в сигнатуре конструктора описывается параметр типа `java.lang.String` и возвращаемое значение типа `void`.

12.6.4. Альтернативные вызовы методов

В прикладном интерфейсе JNI существует несколько вариантов функций, позволяющих вызывать методы на Java из платформенно-ориентированного кода.

И хотя эти функции играют не такую важную роль, как рассмотренные ранее, они могут иногда принести большую пользу.

В частности, функции `CallNonvirtualXxxMethod()` принимают неявный параметр, идентификатор метода, объект класса (который должен соответствовать суперклассу задаваемого неявным образом аргумента), а также явные параметры. Они вызывают версию метода в указанном классе в обход обычного механизма динамической диспетчеризации. Все подобные функции имеют варианты с суффиксом `A` или `V`. Они получают явно заданные параметры в массиве или структуре типа `va_list`, определенной в заголовочном файле `stdarg.h` на C.

Листинг 12.14. Исходный код из файла **printf3/Printf3Test.java**

```
1  import java.io.*;
2
3  /**
4   * @version 1.11 2018-05-01
5   * @author Cay Horstmann
6   */
7  class Printf3Test
8  {
9    public static void main(String[] args)
10   {
11     double price = 44.95;
12     double tax = 7.75;
13     double amountDue = price * (1 + tax / 100);
14     var out = new PrintWriter(System.out);
15     Printf3.fprint(out, "Amount due = %8.2f\n", amountDue);
16     out.flush();
17   }
18 }
```

Листинг 12.15. Исходный код из файла **printf3/Printf3.java**

```
1  import java.io.*;
2  /**
3   * @version 1.10 1997-07-01
4   * @author Cay Horstmann
5  */
6  class Printf3
7  {
8    public static native void fprint(
9           PrintWriter out, String format, double x);
10
11   static
12   {
13     System.loadLibrary("Printf3");
14   }
15 }
```

Листинг 12.16. Исходный код из файла **printf3/Printf3.c**

```
1  /**
2   * @version 1.10 1997-07-01
```

```
3   * @author Cay Horstmann
4   */
5  #include "Printf3.h"
6  #include <string.h>
7  #include <stdlib.h>
8  #include <float.h>
9  /**
10  * @param format Символьная строка со спецификатором
11  * формата для функции printf(), например "%8.2f".
12  * Подстроки "%%" пропускаются
13  * @return Возвращает указатель на спецификатор формата,
14  *         пропуская символ '%', или значение NULL в
15  *         отсутствие однозначного спецификатора формата
16  */
17 char* find_format(const char format[])
18 {
19   char* p;
20   char* q;
21
22   p = strchr(format, '%');
23   while (p != NULL && *(p + 1) == '%')
24          /* пропустить подстроку "%%" */
25     p = strchr(p + 2, '%');
26   if (p == NULL) return NULL;
27       /* проверить единственность символа "%" */
28   p++;
29   q = strchr(p, '%');
30   while (q != NULL && *(q + 1) == '%')
31        /* пропустить подстроку "%%" */
32     q = strchr(q + 2, '%');
33   if (q != NULL)
34     return NULL; /* символ "%" не единственный */
35   q = p + strspn(p, " -0+#"); /* пропустить признаки */
36   q += strspn(q, "0123456789");
37        /* пропустить ширину поля */
38   if (*q == '.') { q++; q += strspn(q, "0123456789"); }
39       /* пропустить точность */
40   if (strchr("eEfFgG", *q) == NULL) return NULL;
41       /* это не формат с плавающей точкой */
42   return p;
43 }
44
45 JNIEXPORT void JNICALL Java_Printf3_fprint(JNIEnv* env,
45                        jclass cl, jobject out,
46                        jstring format, jdouble x)
47 {
48   const char* cformat;
49   char* fmt;
50   jstring str;
51   jclass class_PrintWriter;
52   jmethodID id_print;
53   cformat = (*env)->GetStringUTFChars(env, format, NULL);
54   fmt = find_format(cformat);
55   if (fmt == NULL)
56     str = format;
57   else
58   {
```

```
59      char* cstr;
60      int width = atoi(fmt);
61      if (width == 0) width = DBL_DIG + 10;
62      cstr = (char*) malloc(strlen(cformat) + width);
63      sprintf(cstr, cformat, x);
64      str = (*env)->NewStringUTF(env, cstr);
65      free(cstr);
66    }
67    (*env)->ReleaseStringUTFChars(env, format, cformat);
68
69    /* вызвать метод ps.print(str) */
70
71    /* получить класс */
72    class_PrintWriter = (*env)->GetObjectClass(env, out);
73
74    /* получить идентификатор метода */
75    id_print = (*env)->GetMethodID(env, class_PrintWriter,
76                "print", "(Ljava/lang/String;)V");
77
78    /* вызвать метод */
79    (*env)->CallVoidMethod(env, out, id_print, str);
80  }
```

Вызов методов на Java

- **jmethodID GetMethodID(JNIEnv*** ***env*****, jclass** ***cl*****, const char** ***name*****[], const char** ***methodSignature*****[])**

  Возвращает идентификатор метода в классе.

- ***Xxx*** **CallXxxMethod(JNIEnv*** ***env*****, jobject** ***obj*****, jmethodID** ***id*****,** ***args*****)**
- ***Xxx*** **CallXxxMethodA(JNIEnv*** ***env*****, jobject** ***obj*****, jmethodID** ***id*****, jvalue** ***args*****[])**
- ***Xxx*** **CallXxxMethodV(JNIEnv*** ***env*****, jobject** ***obj*****, jmethodID** ***id*****, va_list** ***args*****)**

  Вызывают метод. В качестве возвращаемого типа ***Xxx*** может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**. Первая функция принимает переменное количество аргументов, присоединяемых после идентификатора метода. Вторая функция получает параметры метода в виде массива значений типа **jvalue**, который имеет вид следующего объединения:

```
typedef union jvalue
{
  jboolean z;
  jbyte b;
  jchar c;
  jshort s;
  jint i;
  jlong j;
  jfloat f;
  jdouble d;
  jobject l;
} jvalue;
```

  Третья функция получает параметры метода в виде структуры **va_list**, определяемой в заголовочном файле **stdarg.h** на C.

Вызов методов на Java *(окончание)*

- ***Xxx*** **CallNonvirtual*Xxx*Method(JNIEnv*** ***env*****, jobject** ***obj*****, jclass** ***cl*****, jmethodID** ***id*****,** ***args*****)**
- ***Xxx*** **CallNonvirtual*Xxx*MethodA(JNIEnv*** ***env*****, jobject** ***obj*****, jclass** ***cl*****, jmethodID** ***id*****, jvalue** ***args*****[])**
- ***Xxx*** **CallNonvirtual*Xxx*MethodV(JNIEnv*** ***env*****, jobject** ***obj*****, jclass** ***cl*****, jmethodID** ***id*****, va_list** ***args*****)**

  Вызывают метод в обход механизма динамической диспетчеризации. В качестве возвращаемого типа ***Xxx*** может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**. Первая функция имеет переменное количество аргументов, присоединяемых после идентификатора метода. Параметры метода передаются второй функции в массиве типа **jvalue**. Третья функция получает параметры в виде структуры **va_list**, определяемой в заголовочном файле **stdarg.h** на C.
- **jmethodID GetStaticMethodID(JNIEnv*** ***env*****, jclass** ***cl*****, const char** ***name*****[], const char** ***methodSignature*****[])**

  Возвращает идентификатор статического метода в классе.
- ***Xxx*** **CallStatic*Xxx*Method(JNIEnv*** ***env*****, jclass** ***cl*****, jmethodID** ***id*****,** ***args*****)**
- ***Xxx*** **CallStatic*Xxx*MethodA(JNIEnv*** ***env*****, jclass** ***cl*****, jmethodID** ***id*****, jvalue** ***args*****[])**
- ***Xxx*** **CallStatic*Xxx*MethodV(JNIEnv*** ***env*****, jclass** ***cl*****, jmethodID** ***id*****, va_list** ***args*****)**

  Вызывают статический метод. В качестве возвращаемого типа ***Xxx*** может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**. Первая функция имеет переменное количество аргументов, присоединяемых после идентификатора метода. Вторая функция получает параметры метода в виде массива типа **jvalue**, третья — в виде структуры **va_list**, определяемой в заголовочном файле **stdarg.h** на C.
- **jobject NewObject(JNIEnv*** ***env*****, jclass** ***cl*****, jmethodID** ***id*****,** ***args*****)**
- **jobject NewObjectA(JNIEnv*** ***env*****, jclass** ***cl*****, jmethodID** ***id*****, jvalue** ***args*****[])**
- **jobject NewObjectV(JNIEnv*** ***env*****, jclass** ***cl*****, jmethodID** ***id*****, va_list** ***args*****)**

  Вызывают конструктор класса. Идентификатор метода получается в результате вызова функции **GetMethodID()** с именем метода в виде строки **"<init>"** и возвращаемым типом **void**. Первая функция имеет переменное количество аргументов, присоединяемых после идентификатора метода. Вторая функция получает параметры метода в виде массива типа **jvalue**, третья — в виде структуры **va_list**, определяемой в заголовочном файле **stdarg.h** на C.

12.7. Доступ к элементам массивов

Все типы массивов в Java имеют соответствующие им типы массивов в C, перечисленные в табл. 12.2.

Таблица 12.2. Соответствие типов массивов в Java и C

Java	C
`boolean[]`	`jbooleanArray`
`byte[]`	`jbyteArray`
`char[]`	`jcharArray`
`int[]`	`jintArray`
`short[]`	`jshortArray`
`long[]`	`jlongArray`
`float[]`	`jfloatArray`
`double[]`	`jdoubleArray`
`Object[]`	`jobjectArray`

НА ЗАМЕТКУ C++! В языке C все типы массивов, по существу, являются синонимами типа **`jobject`**. В языке C++ они упорядочены в иерархическую структуру, как показано на рис. 12.3. Тип **`jarray`** обозначает обобщенный массив.

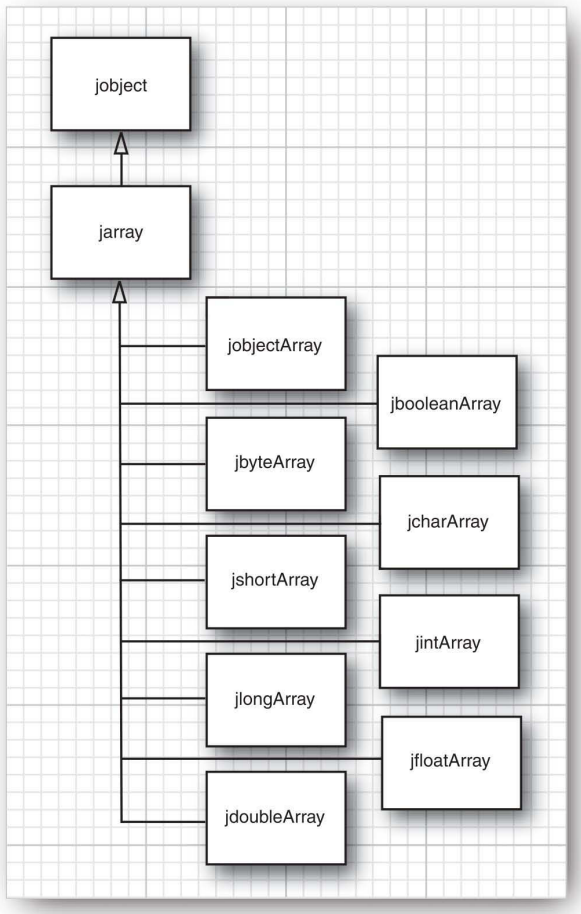


Рис. 12.3. Иерархия наследования типов массивов в C++

Функция `GetArrayLength()` возвращает длину массива, как показано ниже.

```
jarray array = . . .;
jsize length = (*env)->GetArrayLength(env, array);
```

Порядок доступа к элементам массива зависит от того, хранятся ли в нем объекты или значения примитивных типов (числовые, символьные или логические). Для доступа к элементам массива объектов вызываются функции `GetObjectArrayElement()` и `SetObjectArrayElement()`:

```
jobjectArray array = . . .;
int i, j;
jobject x = (*env)->GetObjectArrayElement(env, array, i);
(*env)->SetObjectArrayElement(env, array, j, x);
```

Несмотря на всю свою простоту, такой подход крайне неэффективен. Намного лучше получить непосредственный доступ к элементам массива. И это особенно важно для выполнения операций с векторами или матрицами.

Функция `Get`*`Xxx`*`ArrayElements()` возвращает указатель C на начальный элемент массива. Как и при обращении с обычными символьными строками, если этот указатель больше не нужен, следует вызывать функцию `Release`*`Xxx`*`ArrayElements()`, где *Xxx* — примитивный тип данных, а не объект. Используя указатель C, можно непосредственно считывать и записывать значения элементов массива. Но такой указатель может указывать на *копию* массива, поэтому любые изменения будут отражены в исходном массиве только после вызова функции `Release`*`Xxx`*`ArrayElements()`.

НА ЗАМЕТКУ! Чтобы выяснить, с каким именно массивом приходится иметь дело, с оригиналом или копией, укажите в качестве третьего параметра функции **`Get`*`Xxx`*`ArrayElements()`** переменную типа **`jboolean`**. Если используется копия массива, то в результате вызова данной функции переменная будет содержать значение **`JNI_TRUE`**. Если же вас это мало интересует, укажите в качестве третьего параметра функции **`Get`*`Xxx`*`ArrayElements()`** значение **`NULL`**.

Ниже приведен фрагмент кода, в котором все элементы массива типа `double` умножаются на постоянное значение `scaleFactor` типа `double`. Доступ к отдельным элементам массива `a[i]` осуществляется по указателю C.

```
jdoubleArray array_a = . . .;
double scaleFactor = . . .;
double* a = (*env)->GetDoubleArrayElements(env, array_a, NULL);
for (i = 0; i < (*env)->GetArrayLength(env, array_a); i++)
    a[i] = a[i] * scaleFactor;
(*env)->ReleaseDoubleArrayElements(env, array_a, a, 0);
```

Будет ли виртуальная машина Java копировать этот массив, зависит от способа выделения памяти и организации сборки "мусора". Некоторые копирующие системы сборки "мусора" перемещают объекты и обновляют ссылки на объекты. Такая методика не совместима с "закреплением" за массивом определенного места в памяти, поскольку система сборки "мусора" не может обновлять значения указателей в платформенно-ориентированном коде.

НА ЗАМЕТКУ! В виртуальной машине Oracle JVM массив значений типа **boolean** представляется в виде массива, содержащего 32-разрядные значения. Функция **GetBooleanArrayElements()** распаковывает его и копирует в массив типа **jboolean**.

Для доступа лишь к нескольким элементам крупного массива служат функции `Get`*`Xxx`*`ArrayRegion()` и `Set`*`Xxx`*`ArrayRegion()`. Они копируют из массива Java в массив C и обратно элементы в указанных пределах изменения индексов.

Новые массивы Java в платформенно-ориентированных методах создаются с помощью функции `New`*`Xxx`*`Array()`. Для создания нового массива объектов следует указать его длину, тип элементов массива, а также исходное значение всех элементов (как правило, `NULL`). В приведенном ниже примере кода показано, как это делается.

```
jclass class_Employee = (*env)->FindClass(env, "Employee");
jobjectArray array_e = (*env)->NewObjectArray(env, 100,
                        class_Employee, NULL);
```

Массивы элементов простых типов устроены проще. Поэтому для них достаточно указать длину массива. Так, в результате выполнения следующей строки кода массив заполняется нулями:

```
jdoubleArray array_d = (*env)->NewDoubleArray(env, 100);
```

НА ЗАМЕТКУ! Для работы с прямыми буферами в прикладном интерфейсе JNI имеются следующие три функции:

```
jobject NewDirectByteBuffer(JNIEnv* env, void* address,
                            jlong capacity)
void* GetDirectBufferAddress(JNIEnv* env, jobject buf)
jlong GetDirectBufferCapacity(JNIEnv* env, jobject buf)
```

Прямые буфера применяются в пакете **java.nio** для поддержки более эффективных операций ввода-вывода и сведения к минимуму числа операций копирования данных между платформенно-ориентированным кодом и массивами в Java.

Манипулирование массивами Java

- **jsize GetArrayLength(JNIEnv*** ***env*****, jarray** ***array*****)**

  Возвращает количество элементов в массиве.

- **jobject GetObjectArrayElement(JNIEnv*** ***env*****, jobjectArray** ***array*****, jsize** ***index*****)**

  Возвращает значение указанного элемента массива.

- **void SetObjectArrayElement(JNIEnv*** ***env*****, jobjectArray** ***array*****, jsize** ***index*****, jobject** ***value*****)**

  Устанавливает новое значение в элементе массива.

Манипулирование массивами Java *(окончание)*

- ***Xxx** Get*Xxx*ArrayElements(JNIEnv* *env*, jarray *array*, jboolean* *isCopy*)**

  Выдает указатель C на элементы массива Java. В качестве типа ***Xxx*** поля может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**. Если указатель больше не требуется, он должен быть передан функции **Release*Xxx*ArrayElements()**. Параметр ***isCopy*** принимает значение **NULL** или указатель на переменную типа **jboolean**, которая заполняется значением **JNI_TRUE**, если создается копия массива, а иначе — значение **JNI_FALSE**.

- **void ReleaseXxxArrayElements(JNIEnv* *env*, jarray *array*, *Xxx elems*[], jint *mode*)**

  Уведомляет виртуальную машину Java о том, что указатель, полученный с помощью функции **Get*Xxx*ArrayElements()**, больше не требуется. Параметр ***mode*** может принимать одно из следующих значений: **0** (освободить буфер **elems** после обновления элементов массива), **JNI_COMMIT** (не освобождать буфер **elems** после обновления элементов массива) или **JNI_ABORT** (освободить буфер **elems** без обновления элементов массива).

- **void GetXxxArrayRegion(JNIEnv* *env*, jarray *array*, jint *start*, jint *length*, *Xxx elems*[])**

  Копирует элементы из массива Java в массив C. В качестве типа ***Xxx*** поля может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**.

- **void SetXxxArrayRegion(JNIEnv* *env*, jarray *array*, jint *start*, jint *length*, *Xxx elems*[])**

  Копирует элементы из массива C в массив Java. В качестве типа ***Xxx*** поля может быть указано одно из следующих обозначений: **Object**, **Boolean**, **Byte**, **Char**, **Short**, **Int**, **Long**, **Float** или **Double**.

12.8. Обработка ошибок

Применение платформенно-ориентированных методов представляет существенную угрозу нарушения безопасности программ на Java. В языке C не предусмотрена защита от ошибочного указания диапазона массивов, использования некорректных указателей и т.д. Для сохранения целостности программы и виртуальной машины программистам, создающим платформенно-ориентированные методы, следует уделять особое внимание обработке подобных ошибок. Так, если платформенно-ориентированный метод обнаружит какую-нибудь ошибку, которую он не в состоянии обработать самостоятельно, он должен сообщить о ней виртуальной машине Java, которая сгенерирует соответствующее исключение.

Для создания нового объекта исключения, как правило, вызывается функция `Throw()` или `ThrowNew()`, генерирующая исключение в виртуальной машине Java по завершении платформенно-ориентированного метода. Чтобы воспользоваться функцией `Throw()`, следует вызвать функцию `NewObject()` с целью создать экземпляр класса, производного от класса `Throwable`. Например, в приведенном ниже фрагменте кода создается объект типа `EOFException` и генерируется соответствующее исключение.

```
jclass class_EOFException = (*env)->
       FindClass(env, "java/io/EOFException");
jmethodID id_EOFException = (*env)->
     GetMethodID(env, class_EOFException, "<init>", "()V");
        /* идентификатор конструктора без аргументов */
jthrowable obj_exc = (*env)->
     NewObject(env, class_EOFException, id_EOFException);
(*env)->Throw(env, obj_exc);
```

Но на практике удобнее пользоваться функцией `ThrowNew()`. Она также создает объект исключения, исходя из заданного класса и байтовой последовательности в модифицированном формате UTF-8:

```
(*env)->ThrowNew(env, (*env)->
                 FindClass(env, "java/io/EOFException"),
                 "Unexpected end of file");
```

Функции `Throw()` и `ThrowNew()` только передают исключение, но не прерывают поток управления в платформенно-ориентированном методе. Поэтому виртуальная машина Java может генерировать исключение только по завершении платформенно-ориентированного метода. Это означает, что после каждого вызова функции `Throw()` или `ThrowNew()` должен быть указан оператор `return`.

НА ЗАМЕТКУ C++! Если платформенно-ориентированный метод реализуется на C++, в его теле нельзя генерировать объект исключения Java. Теоретически взаимное преобразование исключений в языках C++ и Java возможно, но в настоящее время оно не реализовано. Для генерирования объектов исключений Java в платформенно-ориентированном коде на C++ придется воспользоваться функциями **`Throw()`** и **`ThrowNew()`**, а также принять меры, чтобы платформенно-ориентированные методы не генерировали исключения C++.

Как правило, платформенно-ориентированный код не должен заниматься обработкой исключений Java, но подобная ситуация может возникнуть при вызове из платформенно-ориентированной функции метода на Java. Например, функция `SetObjectArrayElement()` генерирует исключение типа `ArrayIndexOutOfBoundsException`, если значение индекса выходит за границы массива, а также исключение типа `ArrayStoreException`, если класс хранимого объекта не является производным от класса элемента массива. В подобных случаях в платформенно-ориентированном методе следует вызвать функцию `ExceptionOccured()`, чтобы определить факт генерирования исключения. Так, в результате приведенного ниже вызова возвращается ссылка на текущий объект исключения. Если же в очереди отсутствуют исключения, ожидающие обработки, то возвращается пустое значение `NULL`.

```
jthrowable obj_exc = (*env)->ExceptionOccurred(env);
```

Если требуется лишь проверить, было ли сгенерировано исключение, не получая ссылку на объект исключения, достаточно сделать следующий вызов:

```
jboolean occurred = (*env)->ExceptionCheck(env);
```

В очередном рассматриваемом здесь примере программы реализуется платформенно-ориентированный метод `fprintf()` с некоторыми чрезмерными, на первый взгляд, но весьма полезными на практике мерами для обработки следующих исключений.

- `NullPointerException`, если указатель на форматирующую строку принимает значение `NULL`.
- `IllegalArgumentException`, если форматирующая строка не содержит спецификатор `%`, необходимый для вывода значений типа `double`.
- `OutOfMemoryError`, если произошла ошибка при выполнении функции `malloc()`.

Наконец, чтобы продемонстрировать, каким образом выполняется проверка на наличие исключения при вызове метода на Java из платформенно-ориентированного кода, строка направляется в поток вывода посимвольно, а после вывода каждого символа вызывается функция `ExceptionOccured()`. В листинге 12.17 представлен исходный код платформенно-ориентированного метода, а в листинге 12.18 — исходный код класса, содержащего этот метод. Однако если при вызове метода `PrintWriter.print()` возникает исключение, то платформенно-ориентированный метод завершает свою работу не сразу — сначала он освобождает буфер `cstr`. Когда же происходит возврат из платформенно-ориентированного метода, виртуальная машина Java снова генерирует исключение. Тестовая программа из листинга 12.19 наглядно демонстрирует, каким образом платформенно-ориентированный метод генерирует исключение, если форматирующая строка оказывается недействительной.

Листинг 12.17. Исходный ход из файла **`printf4/Printf4.c`**

```
/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */

#include "Printf4.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>

/**
 * @param format Символьная строка со спецификатором
 * формата для функции printf(), например, "%8.2f".
 * Подстроки "%%" пропускаются
 * @return Возвращает указатель на спецификатор формата,
 *         пропуская символ '%', или значение NULL в
 *         отсутствие однозначного спецификатора формата
*/
char* find_format(const char format[])
{
  char* p;
  char* q;

  p = strchr(format, '%');
  while (p != NULL && *(p + 1) == '%')
        /* пропустить подстроку "%%" */
    p = strchr(p + 2, '%');
  if (p == NULL) return NULL;
```

```
29    /* проверить единственность символа "%" */
30    p++;
31    q = strchr(p, '%');
32    while (q != NULL && *(q + 1) == '%')
33          /* пропустить подстроку "%%" */
34      q = strchr(q + 2, '%');
35    if (q != NULL)
36      return NULL; /* символ "%" не единственный */
37    q = p + strspn(p, " -0+#"); /* пропустить признаки */
38    q += strspn(q, "0123456789");
39        /* пропустить ширину поля */
40    if (*q == '.') { q++; q += strspn(q, "0123456789"); }
41       /* пропустить точность */
42    if (strchr("eEfFgG", *q) == NULL) return NULL;
43       /* это не формат с плавающей точкой */
44    return p;
45  }
46
47  JNIEXPORT void JNICALL Java_Printf4_fprint(JNIEnv* env,
48                         jclass cl, jobject out,
49                         jstring format, jdouble x)
50  {
51    const char* cformat;
52    char* fmt;
53    jclass class_PrintWriter;
54    jmethodID id_print;
55    char* cstr;
56    int width;
57    int i;
58
59    if (format == NULL)
60    {
61      (*env)->ThrowNew(env, (*env)->FindClass(env,
62                       "java/lang/NullPointerException"),
63                      "Printf4.fprint: format is null");
64      return;
65    }
66
67    cformat = (*env)->
68               GetStringUTFChars(env, format, NULL);
69    fmt = find_format(cformat);
70    if (fmt == NULL)
71    {
72      (*env)->ThrowNew(env, (*env)->FindClass(env,
73              "java/lang/IllegalArgumentException"),
74              "Printf4.fprint: format is invalid");
75      return;
76    }
77    width = atoi(fmt);
78    if (width == 0) width = DBL_DIG + 10;
79    cstr = (char*)malloc(strlen(cformat) + width);
80
81    if (cstr == NULL)
82    {
83      (*env)->ThrowNew(env, (*env)->FindClass(env,
84                       "java/lang/OutOfMemoryError"),
```

```
85                      "Printf4.fprint: malloc failed");
86      return;
87    }
88
89    sprintf(cstr, cformat, x);
90
91    (*env)->ReleaseStringUTFChars(env, format, cformat);
92
93    /* вызвать функцию ps.print(str) */
94
95    /* получить класс */
96    class_PrintWriter = (*env)->GetObjectClass(env, out);
97
98    /* получить идентификатор метода */
99    id_print = (*env)->GetMethodID(env, class_PrintWriter,
100                                   "print", "(C)V");
101
102   /* вызвать метод */
103   for (i = 0; cstr[i] != 0
104        && !(*env)-> ExceptionOccurred(env); i++)
105     (*env)->CallVoidMethod(env, out, id_print, cstr[i]);
106
107   free(cstr);
108 }
```

Листинг 12.18. Исходный ход из файла **printf4/Printf4.java**

```
1  import java.io.*;
2
3  /**
4   * @version 1.10 1997-07-01
5   * @author Cay Horstmann
6   */
7  class Printf4
8  {
9    public static native void fprint(
10          PrintWriter ps, String format, double x);
11
12   static
13   {
14     System.loadLibrary("Printf4");
15   }
16 }
```

Листинг 12.19. Исходный ход из файла **printf4/Printf4Test.java**

```
1  import java.io.*;
2
3  /**
4   * @version 1.11 2018-05-01
5   * @author Cay Horstmann
6   */
7  class Printf4Test
8  {
```

```
9     public static void main(String[] args)
10    {
11      double price = 44.95;
12      double tax = 7.75;
13      double amountDue = price * (1 + tax / 100);
14      var out = new PrintWriter(System.out);
15      /* Этот вызов приведет к генерированию
16       * исключения, если в форматирующей строке
17       * отсутствует подстрока "%%"
18       */
19      Printf4.fprint(out, "Amount due = %%8.2f\n", amountDue);
20      out.flush();
21    }
22 }
```

Обработка исключений в Java

- **`jint Throw(JNIEnv* env, jthrowable obj)`**
  Подготавливает исключение, которое должно генерироваться после выхода из платформенно-ориентированного кода. При удачном исходе возвращает нулевое значение, а иначе — отрицательное.
- **`jint ThrowNew(JNIEnv* env, jclass cl, const char msg[])`**
  Подготавливает исключение типа ***`cl`***, которое должно генерироваться при выходе из платформенно-ориентированного кода. При удачном исходе возвращает нулевое значение, а иначе — отрицательное. Параметр ***`msg`*** служит для построения объекта исключения типа **`String`** в виде последовательности байтов в модифицированном формате UTF-8.
- **`jthrowable ExceptionOccurred(JNIEnv* env)`**
  Возвращает объект исключения, если исключение ожидает своей очереди, а иначе — значение **`NULL`**.
- **`jboolean ExceptionCheck(JNIEnv* env)`**
  Возвращает логическое значение **`true`**, если исключение все еще ожидает своей очереди.
- **`void ExceptionClear(JNIEnv* env)`**
  Удаляет все исключения, ожидающие своей очереди.

12.9. Применение прикладного интерфейса API для вызовов

До сих пор рассматривались только программы на Java, делавшие вызовы на C по двум наиболее вероятным причинам: код на C мог выполняться быстрее или требовался доступ к таким функциональным возможностям, которые были недоступны на платформе Java. А теперь рассмотрим обратную ситуацию, когда имеется программа на C или C++ и требуется организовать из нее несколько вызовов кода на Java. Внедрять виртуальную машину Java в программу на C или C++ позволяет прикладной интерфейс API, специально предназначенный для *вызовов*. Ниже приведен минимальный фрагмент кода, который требуется для инициализации виртуальной машины Java.

```
JavaVMOption options[1];
JavaVMInitArgs vm_args;
JavaVM *jvm;
JNIEnv *env;

options[0].optionString = "-Djava.class.path=.";

memset(&vm_args, 0, sizeof(vm_args));
vm_args.version = JNI_VERSION_1_2;
vm_args.nOptions = 1;
vm_args.options = options;

JNI_CreateJavaVM(&jvm, (void**) &env, &vm_args);
```

Вызов функции `JNI_CreateJavaVM()` приводит к созданию виртуальной машины и заполнению указателя `jvm` на нее, а также указателя `env` на среду выполнения. Количество параметров для виртуальной машины Java можно указывать любым, увеличивая размер массива `options` и значение в поле `vm_args.nOptions`. Например, выполнение следующей строки кода приведет к отключению динамического компилятора:

```
options[i].optionString = "-Djava.compiler=NONE";
```

СОВЕТ. Если возникают неполадки следующего характера: программа завершается аварийно, отказывается инициализировать виртуальную машину Java или не в состоянии загрузить классы, необходимо включить режим отладки. Это делается следующим образом:

```
options[i].optionString = "-verbose:jni";
```

В этом режиме выводятся сообщения с подробными сведениями о ходе инициализации виртуальной машины JVM. Если же сообщения о загрузке классов отсутствуют, следует проверить установки как общего пути, так и пути к классам.

После настройки виртуальной машины Java можно приступать к вызову методов на Java способом, описанным в предыдущих разделах, т.е. с помощью указателя `env`. Указатель `jvm` следует использовать только в том случае, если требуется вызвать другие функции из прикладного программного интерфейса API. В настоящее время имеется лишь четыре такие функции. Наиболее важной из них является функция `DestroyJavaVM()`, вызываемая для прекращения работы виртуальной машины Java:

```
(*jvm)->DestroyJavaVM(jvm);
```

К сожалению, в Windows стало трудно осуществлять динамическое связывание с функцией `JNI_CreateJavaVM()` из библиотеки `jre/bin/client/jvm.dll` из-за того, что правила связывания в версии Vista изменились, а компания Oracle по-прежнему делает ставку на устаревшую версию библиотеки рабочих программ на C. В рассматриваемом здесь примере программы данное затруднение разрешается путем загрузки этой библиотеки вручную. Тот же самый подход применяется и в утилите `java`. Убедиться в этом можно, проанализировав исходный код из файла `launcher/java_md.c`, находящегося в архивном файле `src.jar`, входящем в состав JDK.

В листинге 12.20 представлен исходный код программы на C, где сначала инициализируется виртуальная машина Java, а затем вызывается метод `main()` из класса `Welcome`, который рассматривался в главе 2 первого тома настоящего издания. (Перед запуском этой программы следует скомпилировать исходный файл `Welcome.java`.)

Листинг 12.20. Исходный код из файла **invocation/InvocationTest.c**

```
/**
 * @version 1.20 2007-10-26
 * @author Cay Horstmann
*/

#include <jni.h>
#include <stdlib.h>

#ifdef _WINDOWS

#include <windows.h>
static HINSTANCE loadJVMLibrary(void);
typedef jint (JNICALL *CreateJavaVM_t)(JavaVM **,
                      void **, JavaVMInitArgs *);

#endif

int main()
{
  JavaVMOption options[2];
  JavaVMInitArgs vm_args;
  JavaVM *jvm;
  JNIEnv *env;
  long status;

  jclass class_Welcome;
  jclass class_String;
  jobjectArray args;
  jmethodID id_main;

#ifdef _WINDOWS
  HINSTANCE hjvmlib;
  CreateJavaVM_t createJavaVM;
#endif

  options[0].optionString = "-Djava.class.path=.";
  memset(&vm_args, 0, sizeof(vm_args));
  vm_args.version = JNI_VERSION_1_2;
  vm_args.nOptions = 1;
  vm_args.options = options;

#ifdef _WINDOWS
  hjvmlib = loadJVMLibrary();
  createJavaVM = (CreateJavaVM_t) GetProcAddress(
                  hjvmlib, "JNI_CreateJavaVM");
  status = (*createJavaVM)
           (&jvm, (void **) &env, &vm_args);
```

```
#else
  status = JNI_CreateJavaVM(&jvm,
           (void **) &env, &vm_args);
#endif

  if (status == JNI_ERR)
  {
    fprintf(stderr, "Error creating VM\n");
    return 1;
  }

  class_Welcome = (*env)->FindClass(env, "Welcome");
  id_main = (*env)->GetStaticMethodID(
                    env, class_Welcome,
                    "main", "([Ljava/lang/String;)V");

  class_String = (*env)->FindClass(
                   env, "java/lang/String");
  args = (*env)->NewObjectArray(
                  env, 0, class_String, NULL);
  (*env)->CallStaticVoidMethod(env, class_Welcome,
                               id_main, args);
  (*jvm)->DestroyJavaVM(jvm);

  return 0;
}

#ifdef _WINDOWS

static int GetStringFromRegistry(HKEY key,
                         const char *name,
                         char *buf, jint bufsize)
{
  DWORD type, size;

  return RegQueryValueEx(key, name, 0, &type,
                         0, &size) == 0
      && type == REG_SZ
      && size < (unsigned int) bufsize
      && RegQueryValueEx(key, name, 0, 0,
                         buf, &size) == 0;
}

static void GetPublicJREHome(char *buf, jint bufsize)
{
  HKEY key, subkey;
  char version[MAX_PATH];
  /* найти текущую версию JRE */
  char *JRE_KEY =
       "Software\\JavaSoft\\Java Runtime Environment";
  if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, JRE_KEY, 0,
                   KEY_READ, &key) != 0)
  {
    fprintf(stderr, "Error opening registry key '%s'\n",
            JRE_KEY);
    exit(1);
  }
```

```
106
107   if (!GetStringFromRegistry(key, "CurrentVersion",
108       version,  sizeof(version)))
109   {
110     fprintf(stderr, "Failed reading value of registry
111             key: \n\t%s\\CurrentVersion\n", JRE_KEY);
112     RegCloseKey(key);
113     exit(1);
114   }
115
116   /* Найти каталог, где установлена текущая версия */
117   if (RegOpenKeyEx(key, version, 0, KEY_READ,
118                    &subkey) != 0)
119   {
120     fprintf(stderr, "Error opening registry
121             key '%s\\%s'\n", JRE_KEY, version);
122     RegCloseKey(key);
123     exit(1);
124   }
125
126   if (!GetStringFromRegistry(subkey, "JavaHome",
127                              buf, bufsize))
128   {
129     fprintf(stderr, "Failed reading value of registry
130             key: \n\t%s\\%s\\JavaHome\n", 110 JRE_KEY,
131             version);
132     RegCloseKey(key);
133     RegCloseKey(subkey);
134     exit(1);
135   }
136
137   RegCloseKey(key);
138   RegCloseKey(subkey);
139 }
140
141 static HINSTANCE loadJVMLibrary(void)
142 {
143   HINSTANCE h1, h2;
144   char msvcdll[MAX_PATH];
145   char javadll[MAX_PATH];
146   GetPublicJREHome(msvcdll, MAX_PATH);
147   strcpy(javadll, msvcdll);
148   strncat(msvcdll, "\\bin\\msvcr71.dll",
149           MAX_PATH - strlen(msvcdll));
150   msvcdll[MAX_PATH - 1] = '\0';
151   strncat(javadll, "\\bin\\client\\jvm.dll",
152           MAX_PATH - strlen(javadll));
153   javadll[MAX_PATH - 1] = '\0';
154
155   h1 = LoadLibrary(msvcdll);
156   if (h1 == NULL)
157   {
158     fprintf(stderr, "Can't load library msvcr71.dll\n");
159     exit(1);
160   }
161
162   h2 = LoadLibrary(javadll);
```

```
163   if (h2 == NULL)
164   {
165     fprintf(stderr, "Can't load library jvm.dll\n");
166     exit(1);
167   }
168   return h2;
169 }
170
171 #endif
```

Чтобы скомпилировать эту программу в ОС Linux, выполните следующую команду:

```
gcc -I jdk/include -I jdk/include/linux -o InvocationTest \
    -L jdk/jre/lib/i386/client -ljvm InvocationTest.c
```

Если же требуется скомпилировать данную программу в ОС Windows компилятором от корпорации Microsoft, выполните приведенную ниже команду.

```
cl -D_WINDOWS -I jdk\include \
   -I jdk\include\win32 InvocationTest.c \
   jdk\lib\jvm.lib advapi32.lib
```

Непременно убедитесь, что в переменные окружения `INCLUDE LIB` включен путь к заголовочным файлам и файлам библиотек из прикладного интерфейса API для Windows. В среде Cygwin данную программу можно скомпилировать по следующей команде:

```
gcc -D_WINDOWS -mno-cygwin -I jdk\include \
    -I jdk\include\win32 -D__int64="long long" \
    -I c:\cygwin\usr\include\w32api -o InvocationTest
```

Перед запуском данной программы на выполнение в ОС Linux/UNIX следует удостовериться, что в переменную окружения `LD_LIBRARY_PATH` включены каталоги для разделяемых библиотек. Так, если в Linux используется командная оболочка bash, то для этой цели следует выполнить такую команду:

```
export LD_LIBRARY_PATH= \
       jdk/jre/lib/i386/client:$LD_LIBRARY_PATH
```

Функции из прикладного интерфейса API для вызовов

- **`jint JNI_CreateJavaVM(JavaVM** p_jvm, void** p_env, JavaVMInitArgs* vm_args)`**

  Инициализирует виртуальную машину Java. При удачном завершении возвращает нулевое значение, а иначе — значение **`JNI_ERR`**.

- **`jint DestroyJavaVM(JavaVM* jvm)`**

  Удаляет виртуальную машину Java. При удачном завершении возвращает нулевое значение, а иначе — отрицательное. Эту функцию следует вызывать по указателю на виртуальную машину, например:

  `(*jvm)->DestroyJavaVM(jvm)`.

12.10. Практический пример обращения к реестру Windows

В этом разделе рассматривается пример применения всех перечисленных ранее в этой главе способов обращения с платформенно-ориентированными методами для манипулирования символьными строками, массивами и объектами, вызова конструкторов и обработки исключений. Основная цель данного примера — продемонстрировать создание на платформе Java оболочки для подмножества функций, доступных в прикладном интерфейсе API, реализованных на C и предназначенных для работы с реестром ОС Windows. В связи с тем что в рассматриваемом здесь примере программы применяются функции для манипулирования таким специфическим объектом, как реестр Windows, она, разумеется, не может быть перенесена на другие платформы. Именно поэтому в Java не предусмотрено никаких инструментальных средств для работы с реестром, а предполагается, что для этой цели будет использован платформенно-ориентированный код.

12.10.1. Общее представление о реестре Windows

Как известно, реестр представляет собой хранилище конфигурационных данных для операционной системы Windows и прикладных программ. Наличие такого хранилища очень удобно для администрирования, создания резервных копий и настройки прикладных программ. Недостаток реестра, однако, заключается в том, что он является также единой точкой отказа: при наличии ошибок в реестре вся операционная система будет работать со сбоями, а возможно, и вообще не загрузится!

В связи с этим пользоваться реестром для хранения конфигурационных параметров прикладных программ на Java не рекомендуется. С этой целью лучше воспользоваться прикладным интерфейсом API для сохранения глобальных параметров настройки (подробнее об этом см. в главе 10 первого тома настоящего издания). В данном случае реестр используется лишь для того, чтобы продемонстрировать, каким образом платформенно-ориентированные функции из нетривиального прикладного интерфейса API заключаются в оболочку класса Java.

Основным инструментальным средством для работы с реестром является *редактор реестра*. Его применение чревато крупными неприятностями для начинающих пользователей, поэтому для его запуска не предусмотрено никаких визуальных элементов пользовательского интерфейса. Чтобы вызвать редактор реестра из командной строки или в диалоговом окне, которое появляется на экране после выбора пункта меню Пуск⇨Выполнить, следует ввести команду `regedit`. На рис. 12.4 показано, как выглядит окно редактора реестра.

В левой части окна редактора реестра представлены ключи, упорядоченные в виде иерархической древовидной структуры. Следует, однако, иметь в виду, что каждый ключ находится в поддереве, начинающемся в одном из перечисленных ниже узлов с префиксом `HKEY`.

```
HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHINE
. . .
```

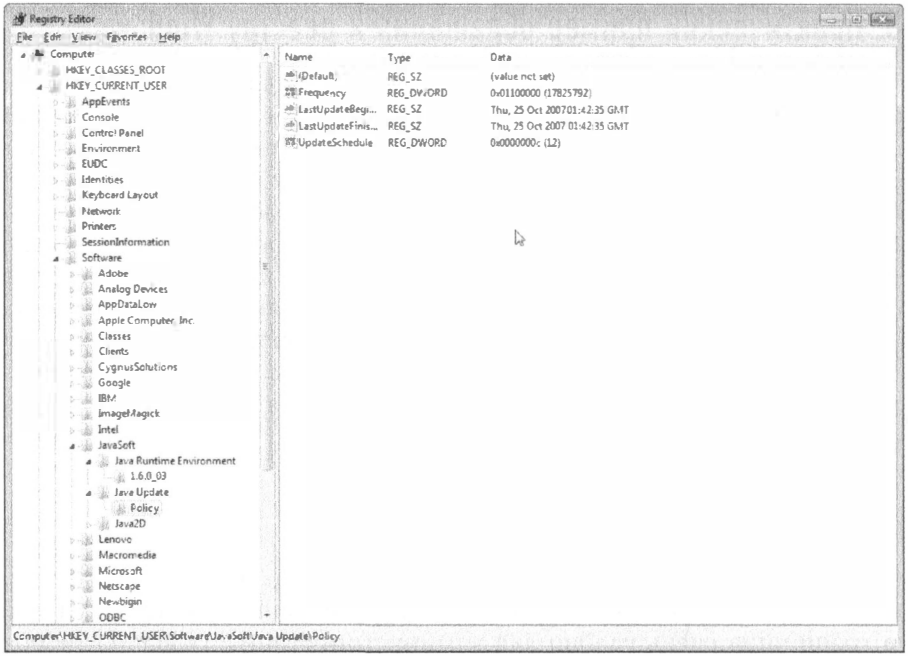


Рис. 12.4. Редактор реестра в Windows

В правой части окна редактора реестра отображаются пары "имя–значение", связанные с конкретным ключом. Так, если установить версию Java 11, то ключ

```
HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Runtime Environment
```

будет содержать такую пару "имя–значение":

```
CurrentVersion="11.0_10"
```

В данном случае значение представлено в виде символьной строки. Но значения могут быть также представлены в виде целых чисел или массивов байтов.

12.10.2. Интерфейс для доступа к реестру на платформе Java

Рассмотрим создание простого интерфейса для организации доступа к реестру из прикладной программы на Java, а также его реализацию в платформенно-ориентированном коде. Этот интерфейс позволяет выполнять лишь несколько операций с реестром. Ради сокращения объема разрабатываемого кода в данном интерфейсе не поддерживаются многие важные операции, в том числе ввод и удаление ключей. Но для их выполнения в таком интерфейсе нетрудно реализовать соответствующие функции.

Используя даже ограниченный набор функций, над реестром можно выполнять следующие действия:

- получать все имена, связанные с ключом;
- читать значение, соответствующее имени;
- связывать с именем новое значение.

Для выполнения подобных действий служит следующий класс Java, инкапсулирующий ключ реестра:

```
public class Win32RegKey
{
  public Win32RegKey(int theRoot, String thePath) { . . . }
  public Enumeration names() { . . . }
  public native Object getValue(String name);
  public native void setValue(String name, Object value);

  public static final int HKEY_CLASSES_ROOT = 0x80000000;
  public static final int HKEY_CURRENT_USER = 0x80000001;
  public static final int HKEY_LOCAL_MACHINE = 0x80000002;
  . . .
}
```

Метод `names()` возвращает объект типа `Enumeration`, который содержит все имена, связанные с данным ключом. Их можно получить с помощью методов `hasMoreElements()`/`nextElement()`. Метод `getValue()` возвращает объект, который может быть символьной строкой, экземпляром класса `Integer` или байтовым массивом. Параметр `value` метода `setValue()` также должен относиться к одному из этих трех ссылочных типов данных.

12.10.3. Реализация функций доступа к реестру в виде платформенно-ориентированных методов

В рассматриваемом здесь примере программы необходимо реализовать следующие действия над реестром Windows:

- получение значения по имени;
- установка значения по имени;
- перечисление имен по ключам.

Ранее в этой главе уже были рассмотрены все основные инструментальные средства, требующиеся для взаимного преобразования символьных строк и массивов на Java и C. Кроме того, ранее было показано, каким образом генерируются и обрабатываются исключения при возникновении ошибок.

Но по сравнению с примерами из предыдущих разделов применение платформенно-ориентированных методов усложняется следующими двумя обстоятельствами. Во-первых, методы `getValue()` и `setValue()` манипулируют данными типа `Object`, которые могут относиться к одному из следующих конкретных типов: `String`, `Integer` или `byte[]`. Кроме того, в объекте перечислимого типа должны храниться сведения о состоянии между последовательными вызовами методов `hasMoreElements()`/`nextElement()`. Рассмотрим сначала метод `getValue()`, выполняющий перечисленные ниже действия (его исходный код представлен в листинге 12.22).

1. Открывает ключ в реестре. В прикладном интерфейсе API для доступа к реестру требуется, чтобы ключи были открытыми.
2. Запрашивает тип и длину значения, связанного с данным именем.
3. Считывает данные запрашиваемого значения в буфер.

4. Вызывает функцию NewStringUTF() для создания новой символьной строки с данными запрашиваемого значения, если это значение относится к типу REG_SZ (т.е. является строкой).
5. Вызывает конструктор класса Integer, если запрашиваемое значение относится к типу REG_DWORD (т.е. является 32-разрядным целочисленным значением).
6. Взывает сначала функцию NewByteArray() для создания нового байтового массива, а затем функцию SetByteArrayRegion() для копирования данных запрашиваемого значения в полученный массив, если это значение относится к типу REG_BINARY (т.е. является байтовым массивом).
7. Если же запрашиваемое значение не относится ни к одному из перечисленных выше типов или если при вызове функции из прикладного интерфейса API возникла какая-нибудь ошибка, генерирует исключение и освобождает все полученные до сих пор ресурсы.
8. Закрывает ключ в реестре и возвращает созданный объект (типа String, Integer или byte[]).

Как видите, рассматриваемый здесь пример наглядно показывает, каким образом формируются объекты разных типов в Java.

В данном платформенно-ориентированном методе совсем не трудно организовать возврат обобщенного значения, поскольку ссылку типа jstring, jint или byte[] можно возвратить просто как ссылку типа jobject. Но метод setValue() получает ссылку на объект типа Object и должен точно определить его конкретный тип, чтобы сохранить его в виде символьной строки, целочисленного значения или байтового массива. Этот тип можно определить, запросив класс объекта value и получив ссылки на классы java.lang.String, java.lang.Integer или java.lang.byte[], а затем сравнив их с помощью функции IsAssignableFrom().

Так, если class1 и class2 являются ссылками на два разных класса, то в результате вызова приведенного ниже метода возвратится значение JNI_TRUE, при условии, что классы class1 и class2 одинаковы или же класс class1 является подклассом, производным от класса class2.

```
(*env)->IsAssignableFrom(env, class1, class2)
```

Но в любом случае ссылки на объекты класса class1 могут быть приведены к типу class2. Так, если приведенный ниже метод возвращает логическое значение true, это означает, что объект value является байтовым массивом.

```
(*env)->IsAssignableFrom(env,
       (*env)->GetObjectClass(env, value),
       (*env)->FindClass(env, "[B"))
```

Теперь рассмотрим метод setValue(), выполняющий перечисленные ниже действия.

1. Открывает ключ в реестре для записи.
2. Определяет тип записываемого значения.
3. Вызывает функцию GetStringUTFChars() для получения указателя на символы, если значение относится к типу String.

4. Вызывает метод `intValue()` для извлечения целочисленного значения, хранящегося в объекте-оболочке, если значение относится к типу `Integer`.
5. Вызывает метод `GetByteArrayElements()` для получения указателя на байты, если значение относится к типу `byte[]`.
6. Передает данные и длину значения в реестр.
7. Закрывает ключ в реестре.
8. Освобождает указатель на данные, если речь идет о значении типа `String` или `byte[]`.

Наконец, рассмотрим платформенно-ориентированные методы для перечисления имен по ключам реестра. Как следует из листинга 12.21, эти методы входят в состав класса `Win32RegKeyNameEnumeration`. Чтобы начать процесс перечисления, следует сначала открыть ключ в реестре. На время существования объекта перечислимого типа необходимо сохранить дескриптор ключа. Этот дескриптор относится к типу `DWORD` (32-разрядное целочисленное значение) и поэтому может быть сохранен в поле `hkey` объекта перечислимого типа. В самом начале процесса перечисления это поле инициализируется с помощью метода `SetIntField()`. В дальнейшем значение поля `hkey` читается с помощью метода `GetIntField()`.

В объекте перечислимого типа из рассматриваемого здесь примера сохраняются также три других элемента данных. В самом начале процесса перечисления в реестре можно запросить количество пар "имя–значение", а также длину самого длинного имени. Эти сведения требуются для выделения символьного массива на C и сохраняются в полях `count` и `maxsize` объекта перечислимого типа. Наконец, поле `index` этого объекта инициализируется значением `-1`, указывая на начало процесса перечисления. После инициализации других полей экземпляра в этом поле устанавливается нулевое значение, которое затем инкрементируется на каждом шаге перечисления.

Рассмотрим по очереди действия платформенно-ориентированных методов, поддерживающих перечисление. Начнем с самого простого метода `hasMoreElements()`, который выполняет следующие действия.

1. Извлекает значения полей `index` и `count`.
2. Если в поле `index` оказывается значение `-1`, то вызывается функция `startNameEnumeration()`, которая открывает соответствующий ключ в реестре, запрашивает значение счетчика и максимальную длину, а затем инициализирует поля `hkey`, `count`, `maxsize` и `index`.
3. Если значение в поле `index` меньше значения в поле `count`, возвращает значение `JNI_TRUE`, а иначе — значение `JNI_FALSE`.

Метод `nextElement()` выполняет более сложные действия, перечисленные ниже.

1. Извлекает значения полей `index` и `count`.
2. Если значение в поле `index` равно `-1`, вызывает функцию `startNameEnumeration()`, которая открывает соответствующий ключ в реестре, запрашивает значение счетчика и максимальную длину, а затем инициализирует поля `hkey`, `count`, `maxsize` и `index`.

3. Если значение в поле `index` равно значению в поле `count`, генерирует исключение типа `NoSuchElementException`.
4. Читает следующее имя из реестра.
5. Инкрементирует значение в поле `index`.
6. Если значение в поле `index` снова оказывается равным значению в поле `count`, закрывает ключ в реестре.

Перед компиляцией исходного кода программы из данного примера не забудьте выполнить команду `javac` с параметром `-h` и обоими исходными файлами — `Win32RegKey` и `Win32RegKeyNameEnumeration`. Для компилятора от корпорации Microsoft соответствующая командная строка будет выглядеть следующим образом:

```
cl -I jdk\include -I jdk\include\win32 -LD Win32RegKey.c \
    advapi32.lib -FeWin32RegKey.dll
```

В среде Cygwin аналогичная команда будет выглядеть так:

```
gcc -mno-cygwin -D __int64="long long" -I jdk\include \
    -I jdk\include\win32 \
    -I c:\cygwin\usr\include\w32api -shared -Wl, \
    --add-stdcall-alias \
    -o Win32RegKey.dll Win32RegKey.c
```

Но прикладной интерфейс API для доступа к реестру рассчитан исключительно на применение в ОС Windows, поэтому в других операционных системах рассматриваемая здесь программа работать не будет.

В листинге 12.23 приведен код программы, позволяющей протестировать новые функции, разработанные для обращения с реестром. Для проведения такого тестирования три пары "имя–значение" (символьная строка, целое число и байтовый массив) сначала добавляются в следующий ключ реестра:

```
HKEY_CURRENT_USER\Software\JavaSoft\Java Runtime Environment
```

Затем перечисляются все имена этого ключа и извлекаются их значения. Ниже приведен результат выполнения данной программы.

```
Default user=Harry Hacker
Lucky number=13
Small primes=2 3 5 7 11 13
```

Несмотря на то что добавление подобных пар имен и значений в данный ключ вряд ли сможет нанести какой-нибудь вред реестру, после выполнения рассматриваемого здесь примера программы их нетрудно удалить, если потребуется, с помощью редактора реестра.

Листинг 12.21. Исходный код из файла **`win32reg/Win32RegKey.java`**

```
1  import java.util.*;
2
3  /**
4   * Объект класса Win32RegKey служит для получения и
5   * установки значений в ключах реестра Windows
6   * @version 1.00 1997-07-01
```

```
7   * @author Cay Horstmann
8   */
9  public class Win32RegKey
10 {
11   public static final int HKEY_CLASSES_ROOT =
12                           0x80000000;
13   public static final int HKEY_CURRENT_USER =
14                           0x80000001;
15   public static final int HKEY_LOCAL_MACHINE =
16                           0x80000002;
17   public static final int HKEY_USERS = 0x80000003;
18   public static final int HKEY_CURRENT_CONFIG =
19                           0x80000005;
20   public static final int HKEY_DYN_DATA = 0x80000006;
21
22   private int root;
23   private String path;
24
25   /**
26    * Получает значение ключа в реестре
27    * @param name Имя ключа
28    * @return Значение, связанное с заданным ключом
29    */
30   public native Object getValue(String name);
31
32   /**
33    * Устанавливает значение ключа в реестре
34    * @param name Имя ключа
35    * @param value Новое значение
36    */
37   public native void setValue(String name, Object value);
38
39   /**
40    * Создает объект ключа в реестре
41    * @param theRoot Один из следующих ключей:
42    *                HKEY_CLASSES_ROOT, HKEY_CURRENT_USER,
43    *                HKEY_LOCAL_MACHINE, HKEY_USERS,
44    *                HKEY_CURRENT_CONFIG, HKEY_DYN_DATA
45    * @param thePath Путь к ключу в реестре
46    */
47   public Win32RegKey(int theRoot, String thePath)
48   {
49     root = theRoot;
50     path = thePath;
51   }
52
53   /**
54    * Перечисляет все имена ключей в реестре по пути,
55    * описываемому в данном объекте
56    * @return Возвращает список всех перечисляемых имен
57   */
58   public Enumeration<String> names()
59   {
60     return new Win32RegKeyNameEnumeration(root, path);
61   }
62   static
```

```
63   {
64     System.loadLibrary("Win32RegKey");
65   }
66 }
67
68 class Win32RegKeyNameEnumeration
69       implements Enumeration<String>
70 {
71   public native String nextElement();
72   public native boolean hasMoreElements();
73   private int root;
74   private String path;
75   private int index = -1;
76   private int hkey = 0;
77   private int maxsize;
78   private int count;
79
80   Win32RegKeyNameEnumeration(int theRoot, String thePath)
81   {
82     root = theRoot;
83     path = thePath;
84   }
85 }
86
87 class Win32RegKeyException extends RuntimeException
88 {
89   public Win32RegKeyException()
90   {
91   }
92
93   public Win32RegKeyException(String why)
94   {
95     super(why);
96   }
97 }
```

Листинг 12.22. Исходный код из файла **win32reg/Win32RegKey.c**

```
1   /**
2    * @version 1.00 1997-07-01
3    * @author Cay Horstmann
4    */
5
6   #include "Win32RegKey.h"
7   #include "Win32RegKeyNameEnumeration.h"
8   #include <string.h>
9   #include <stdlib.h>
10  #include <windows.h>
11  JNIEXPORT jobject JNICALL Java_Win32RegKey_getValue(
12            JNIEnv* env, jobject this_obj, jobject name)
13  {
14    const char* cname;
15    jstring path;
16    const char* cpath;
17    HKEY hkey;
```

```
18    DWORD type;
19    DWORD size;
20    jclass this_class;
21    jfieldID id_root;
22    jfieldID id_path;
23    HKEY root;
24    jobject ret;
25    char* cret;
26
27    /* получить класс */
28    this_class = (*env)->GetObjectClass(env, this_obj);
29
30    /* получить идентификаторы полей */
31    id_root = (*env)->GetFieldID(env, this_class,
32                                 "root", "I");
33    id_path = (*env)->GetFieldID(env, this_class, "path",
34                                 "Ljava/lang/String;");
35
36    /* получить поля */
37    root = (HKEY) (*env)->GetIntField(env, this_obj,
38                                      id_root);
39    path = (jstring)(*env)->GetObjectField(env,
40                            this_obj, id_path);
41    cpath = (*env)->GetStringUTFChars(env, path, NULL);
42
43    /* открыть ключ в реестре */
44    if (RegOpenKeyEx(root, cpath, 0,
45                     KEY_READ, &hkey) != ERROR_SUCCESS)
46    {
47      (*env)->ThrowNew(env, (*env)->FindClass(env,
48                             "Win32RegKeyException"),
49                             "Open key failed");
50      (*env)->ReleaseStringUTFChars(env, path, cpath);
51      return NULL;
52    }
53
54    (*env)->ReleaseStringUTFChars(env, path, cpath);
55    cname = (*env)->GetStringUTFChars(env, name, NULL);
56
57    /* обнаружить тип и длину значения */
58    if (RegQueryValueEx(hkey, cname, NULL, &type,
59                        NULL, &size) != ERROR_SUCCESS)
60    {
61      (*env)->ThrowNew(env, (*env)->FindClass(env,
59                             "Win32RegKeyException"),
60                             "Query value key failed");
61    RegCloseKey(hkey);
62      (*env)->ReleaseStringUTFChars(env, name, cname);
63      return NULL;
64    }
65    /* выделить область памяти для хранения
66       конкретного значения */
67    cret = (char*)malloc(size);
68
69    /* прочитать значение */
70    if (RegQueryValueEx(hkey, cname, NULL, &type,
```

```
71                          cret, &size) != ERROR_SUCCESS)
72    {
73      (*env)->ThrowNew(env, (*env)->FindClass(env,
74                              "Win32RegKeyException"),
75                              "Query value key failed");
76      free(cret);
77      RegCloseKey(hkey);
78      (*env)->ReleaseStringUTFChars(env, name, cname);
79      return NULL;
80    }
81
82    /* сохранить значение как символьную строку,
83       целое число или байтовый массив в зависимости
84       от его типа */
85    if (type == REG_SZ)
86    {
87      ret = (*env)->NewStringUTF(env, cret);
88    }
89    else if (type == REG_DWORD)
90    {
91      jclass class_Integer = (*env)->
92             FindClass(env, "java/lang/Integer");
93      /* получить идентификатор метода для конструктора */
94      jmethodID id_Integer = (*env)->GetMethodID(env,
95                class_Integer, "<init>", "(I)V");
96      int value = *(int*) cret;
97      /* вызвать конструктор */
98      ret = (*env)->NewObject(env, class_Integer,
99                              id_Integer, value);
100   }
101   else if (type == REG_BINARY)
102   {
103     ret = (*env)->NewByteArray(env, size);
104     (*env)->SetByteArrayRegion(env,
105             (jarray) ret, 0, size, cret);
106   }
107   else
108   {
109     (*env)->ThrowNew(env, (*env)->FindClass(env,
110                             "Win32RegKeyException"),
111                             "Unsupported value type");
112     ret = NULL;
113   }
114
115   free(cret);
116   RegCloseKey(hkey);
117   (*env)->ReleaseStringUTFChars(env, name, cname);
118
119   return ret;
120 }
121 JNIEXPORT void JNICALL Java_Win32RegKey_setValue(
122                        JNIEnv* env,jobject this_obj,
123                        jstring name, jobject value)
124 {
125   const char* cname;
126   jstring path;
```

```
127   const char* cpath;
128   HKEY hkey;
129   DWORD type;
130   DWORD size;
131   jclass this_class;
132   jclass class_value;
133   jclass class_Integer;
134   jfieldID id_root;
135   jfieldID id_path;
136   HKEY root;
137   const char* cvalue;
138   int ivalue;
139
140   /* получить класс */
141   this_class = (*env)->GetObjectClass(env, this_obj);
142
143   /* получить идентификаторы полей */
144   id_root = (*env)->GetFieldID(env,
145                                this_class, "root", "I");
146   id_path = (*env)->GetFieldID(env, this_class, "path",
147                                "Ljava/lang/String;");
148
149   /* получить поля */
150   root = (HKEY)(*env)->GetIntField(
151                        env, this_obj, id_root);
152   path = (jstring)(*env)->GetObjectField(
153                           env, this_obj, id_path);
154   cpath = (*env)->GetStringUTFChars(env, path, NULL);
155
156   /* открыть ключ в реестре */
157   if (RegOpenKeyEx(root, cpath, 0,
158                    KEY_WRITE, &hkey) != ERROR_SUCCESS)
159   {
160     (*env)->ThrowNew(env, (*env)->FindClass(
161                       env, "Win32RegKeyException"),
162                       "Open key failed");
163     (*env)->ReleaseStringUTFChars(env, path, cpath);
164     return;
165   }
166
167   (*env)->ReleaseStringUTFChars(env, path, cpath);
168   cname = (*env)->GetStringUTFChars(env, name, NULL);
169
170   class_value = (*env)->GetObjectClass(env, value);
171   class_Integer = (*env)->FindClass(
172                           env, "java/lang/Integer");
173   /* определить тип объекта, представляющего значение */
174   if ((*env)->IsAssignableFrom(env, class_value,
175       (*env)-> FindClass(env, "java/lang/String")))
176   {
177     /* это строка, поэтому получить
178        указатель на ее символы */
179     cvalue = (*env)->GetStringUTFChars(
180                       env, (jstring) value, NULL);
181     type = REG_SZ;
182     size = (*env)->GetStringLength(
```

```
183                     env, (jstring) value) + 1;
184   }
185   else if ((*env)->IsAssignableFrom(
186             env, class_value, class_Integer))
187   {
188     /* это целое значение, поэтому вызвать
189        метод intValue(), чтобы получить значение */
190     jmethodID id_intValue = (*env)->GetMethodID(
191               env, class_Integer, "intValue", "()I");
192     ivalue = (*env)->CallIntMethod(
193                env, value, id_intValue);
194     type = REG_DWORD;
195     cvalue = (char*)&ivalue;
196     size = 4;
197   }
198   else if ((*env)->IsAssignableFrom(env, class_value,
199             (*env)->FindClass(env, "[B")))
200   {
201     /* это байтовый массив, поэтому получить
202        указатель на байты */
203     type = REG_BINARY;
204     cvalue = (char*)(*env)->GetByteArrayElements(
205                              env, (jarray) value, NULL);
206     size = (*env)->GetArrayLength(env, (jarray) value);
207   }
208   else
209   {
210     /* неизвестно, как обрабатывать этот тип данных */
211     (*env)->ThrowNew(env, (*env)->FindClass(
212                       env, "Win32RegKeyException"),
213                       "Unsupported value type");
214     RegCloseKey(hkey);
215     (*env)->ReleaseStringUTFChars(env, name, cname);
216     return;
217   }
218
219   /* установить значение */
220   if (RegSetValueEx(hkey, cname, 0, type,
221                     cvalue, size) != ERROR_SUCCESS)
222   {
223     (*env)->ThrowNew(env, (*env)->FindClass(
221                       env, "Win32RegKeyException"),
222                       "Set value failed");
223   }
224
225   RegCloseKey(hkey);
226   (*env)->ReleaseStringUTFChars(env, name, cname);
227
228   /* если значение оказалось символьной строкой или
229    * байтовым массивом, освободить указатель */
230   if (type == REG_SZ)
231   {
232     (*env)->ReleaseStringUTFChars(
233             env, (jstring) value, cvalue);
234   }
235   else if (type == REG_BINARY)
```

```
236   {
237     (*env)->ReleaseByteArrayElements(
238             env, (jarray) value, (jbyte*) cvalue, 0);
239   }
240 }
241
242 /* Вспомогательная функция, начинающая процесс
243    перечисления имен по ключам */
244 static int startNameEnumeration(
245       JNIEnv* env, jobject this_obj, jclass this_class)
246 {
247   jfieldID id_index;
248   jfieldID id_count;
249   jfieldID id_root;
250   jfieldID id_path;
251   jfieldID id_hkey;
252   jfieldID id_maxsize;
253
254   HKEY root;
255   jstring path;
256   const char* cpath;
257   HKEY hkey;
258   DWORD maxsize = 0;
259   DWORD count = 0;
260
261   /* получить идентификаторы полей */
262   id_root = (*env)->GetFieldID(
263                     env, this_class, "root", "I");
264   id_path = (*env)->GetFieldID(env, this_class, "path",
265                                "Ljava/lang/String;");
266   id_hkey = (*env)->GetFieldID(
267                     env, this_class, "hkey", "I");
268   id_maxsize = (*env)->GetFieldID(env, this_class,
269                                   "maxsize", "I");
270   id_index = (*env)->GetFieldID(env, this_class,
271                                 "index", "I");
272   id_count = (*env)->GetFieldID(env, this_class,
273                                 "count", "I");
274
275   /* получить значения из полей */
276   root = (HKEY)(*env)->GetIntField(
277                       env, this_obj, id_root);
278   path = (jstring)(*env)->GetObjectField(
279                         env, this_obj, id_path);
280   cpath = (*env)->GetStringUTFChars(env, path, NULL);
281
282   /* открыть ключ в реестре */
283   if (RegOpenKeyEx(root, cpath, 0,
284                    KEY_READ, &hkey) != ERROR_SUCCESS)
285   {
286     (*env)->ThrowNew(env, (*env)->FindClass(
287                           env, "Win32RegKeyException"),
288                           "Open key failed");
289     (*env)->ReleaseStringUTFChars(env, path, cpath);
290     return -1;
291   }
```

```
292   (*env)->ReleaseStringUTFChars(env, path, cpath);
293   /* запросить число и максимальную длину имен */
294   if (RegQueryInfoKey(hkey, NULL, NULL, NULL, NULL,
295               NULL, NULL, &count, &maxsize,
296               NULL, NULL, NULL) != ERROR_SUCCESS)
297   {
298     (*env)->ThrowNew(env, (*env)->FindClass(env,
299                            "Win32RegKeyException"),
300                            "Query info key failed");
301     RegCloseKey(hkey);
302     return -1;
303   }
304
305   /* установить значения в полях */
306   (*env)->SetIntField(env, this_obj,
307                       id_hkey, (DWORD) hkey);
308   (*env)->SetIntField(env, this_obj,
309                       id_maxsize, maxsize + 1);
310   (*env)->SetIntField(env, this_obj, id_index, 0);
311   (*env)->SetIntField(env, this_obj, id_count, count);
312   return count;
313 }
314
315 JNIEXPORT jboolean JNICALL
316      Java_Win32RegKeyNameEnumeration_hasMoreElements(
317                      JNIEnv* env, jobject this_obj)
318 { jclass this_class;
319   jfieldID id_index;
320   jfieldID id_count;
321   int index;
322   int count;
323   /* получить класс */
324   this_class = (*env)->GetObjectClass(env, this_obj);
325
326   /* получить идентификаторы полей */
327   id_index = (*env)->GetFieldID(
328                       env, this_class, "index", "I");
329   id_count = (*env)->GetFieldID(
330                       env, this_class, "count", "I");
331
332   index = (*env)->GetIntField(env, this_obj, id_index);
333   if (index == -1) /* в первый раз */
334   {
335     count = startNameEnumeration(
336             env, this_obj, this_class);
337     index = 0;
338   }
339   else
340   count = (*env)->GetIntField(env, this_obj, id_count);
341   return index < count;
342 }
343
344 JNIEXPORT jobject JNICALL
345           Java_Win32RegKeyNameEnumeration_nextElement(
346                        JNIEnv* env, jobject this_obj)
347 {
```

```
348   jclass this_class;
349   jfieldID id_index;
350   jfieldID id_hkey;
351   jfieldID id_count;
352   jfieldID id_maxsize;
353
354   HKEY hkey;
355   int index;
356   int count;
357   DWORD maxsize;
358
359   char* cret;
360   jstring ret;
361
362   /* получить класс */
363   this_class = (*env)->GetObjectClass(env, this_obj);
364
365   /* получить идентификаторы полей */
366   id_index = (*env)->GetFieldID(
367                     env, this_class, "index", "I");
368   id_count = (*env)->GetFieldID(
369                     env, this_class, "count", "I");
370   id_hkey = (*env)->GetFieldID(
371                    env, this_class, "hkey", "I");
372   id_maxsize = (*env)->GetFieldID(
373                       env, this_class, "maxsize", "I");
374
375   index = (*env)->GetIntField(env, this_obj, id_index);
376   if (index == -1) /* first time */
377   {
378     count = startNameEnumeration(
379             env, this_obj, this_class);
380     index = 0;
381   }
382   else
383     count = (*env)->GetIntField(
384                    env, this_obj, id_count);
385
386   if (index >= count) /* already at end */
387   {
388     (*env)->ThrowNew(env, (*env)->FindClass(
389             env, "java/util/NoSuchElementException"),
390             "past end of enumeration");
391     return NULL;
392   }
393
394   maxsize = (*env)->GetIntField(
395                    env, this_obj, id_maxsize);
396   hkey = (HKEY)(*env)->GetIntField(
397                       env, this_obj, id_hkey);
398   cret = (char*)malloc(maxsize);
399
400   /* обнаружить следующее имя */
401   if (RegEnumValue(hkey, index, cret, &maxsize, NULL,
402                    NULL, NULL, NULL)
403      != ERROR_SUCCESS)
```

```
404   {
405     (*env)->ThrowNew(env, (*env)->FindClass(env,
406                            "Win32RegKeyException"),
407                            "Enum value failed");
408     free(cret);
409     RegCloseKey(hkey);
410     (*env)->SetIntField(
411             env, this_obj, id_index, count);
412     return NULL;
413   }
414
415   ret = (*env)->NewStringUTF(env, cret);
416   free(cret);
417
418   /* инкрементировать индекс */
419   index++;
420   (*env)->SetIntField(env, this_obj, id_index, index);
421
422   if (index == count) /* at end */
423   {
424     RegCloseKey(hkey);
425   }
426
427   return ret;
428 }
```

Листинг 12.23. Исходный код из файла **win32reg/Win32RegKeyTest.java**

```
1  import java.util.*;
2
3  /**
4   @version 1.03 2018-05-01
5   @author Cay Horstmann
6  */
7  public class Win32RegKeyTest
8  {
9    public static void main(String[] args)
10   {
11     var key = new Win32RegKey(
12         Win32RegKey.HKEY_CURRENT_USER,
13         "Software\\JavaSoft\\Java Runtime Environment");
14
15     key.setValue("Default user", "Harry Hacker");
16     key.setValue("Lucky number", new Integer(13));
17     key.setValue("Small primes",
18                  new byte[] { 2, 3, 5, 7, 11 });
19
20     Enumeration<String> e = key.names();
21
22     while (e.hasMoreElements())
23     {
24       String name = e.nextElement();
25       System.out.print(name + "=");
26
27       Object value = key.getValue(name);
```

```
28
29        if (value instanceof byte[])
30          for (byte b : (byte[]) value)
31            System.out.print((b & 0xFF) + " ");
32        else
33          System.out.print(value);
34
35        System.out.println();
36      }
37    }
38 }
```

Функции запроса типа данных

- **jboolean IsAssignableFrom(JNIEnv* *env*, jclass *cl1*, jclass *cl2*)**

  Возвращает значение **JNI_TRUE**, если объекты одного класса (*cl1*) могут быть присвоены объектам другого класса (*cl2*), а иначе — значение **JNI_FALSE**. Проверяет, являются ли классы *cl1* и *cl2* одинаковыми, является ли класс *cl1* производным от класса *cl2*, представляет ли класс *cl2* интерфейс, реализуемый классом *cl1* или же каким-нибудь из его суперклассов.

- **jclass GetSuperclass(JNIEnv* *env*, jclass *cl*)**

  Возвращает суперкласс указанного класса. Если же *cl* представляет класс **Object** или интерфейс, то возвращается значение **NULL**.

Вот и подошел к концу второй том настоящего издания, завершающий долгий экскурс в Java. В этом томе у вас была возможность ознакомиться со многими расширенными средствами программирования на Java и дополнительными прикладными интерфейсами API. В начале этого тома были рассмотрены такие важные для всякого программирующего на Java темы, как потоки данных, XML, сетевые соединения, базы данных и интернационализация. В завершающих этот том главах обсуждались специальные вопросы безопасности, модуляризации, обработки аннотаций, расширенных средств для построения графики и применения платформенно-ориентированных методов. Надеемся, что этот экскурс в расширенные средства программирования на Java и прикладные интерфейсы API оказался для вас занимательным и полезным, и теперь вы сможете применить приобретенные знания и навыки в своих проектах.

Предметный указатель

Б

В

Д

З

И

K

M

Н

О

П

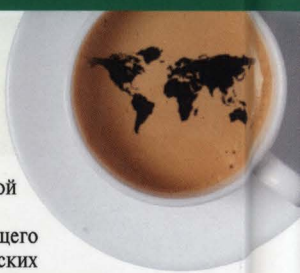
P

У

Ф

Исчерпывающее руководство по разработке пользовательских интерфейсов и корпоративных приложений на Java!

Полностью обновлено по версии Java 11



Это одиннадцатое полностью обновленное по версии Java 11 издание представляет собой солидное справочное руководство, рассчитанное на опытных программистов, стремящихся писать надежный код на Java для реальных приложений. Во втором томе настоящего издания освещаются расширенные средства для построения графических пользовательских интерфейсов, работа в сети, вопросы программирования корпоративных приложений и безопасности, а также новая эффективная модульная система на платформе Java.

В этом надежном и полезном практическом руководстве описаны расширенные языковые средства, библиотеки и прикладные интерфейсы, проиллюстрированные тщательно подобранными и проверенными примерами, отражающими нормы передовой практики и современный стиль программирования на Java. Приведенные примеры просты для понимания, практически полезны и служат неплохой отправной точкой для написания собственного кода.

ОСНОВНЫЕ ТЕМЫ КНИГИ

- Наилучшие методики, идиомы и нормы передовой практики для написания высококачественного и надежного кода на Java.
- Выгодное применение прикладных интерфейсов современной системы ввода-вывода, сериализации объектов и регулярных выражений.
- Эффективное подключение программ на Java к сетевым службам, реализация клиентов и серверов, а также сбор веб-данных.
- Компиляция и выполнение кода через прикладные интерфейсы для сценариев на других языках и компилятора Java, а также обработка аннотаций.
- Подробное изложение модульной системы на платформе Java и перенос кода для работы с ней.
- Эффективное повышение безопасности с помощью современных средств, наиболее ценных для прикладного программирования.
- Программирование развитых пользовательских интерфейсов клиентских приложений и формирование изображений на сервере.
- Синтаксический анализ, проверка достоверности данных, формирование XML-документов, применение XPath, XSL и многих других средств обработки данных формата XML в Java.
- Программирование баз данных средствами JDBC.
- Интернационализация прикладных программ с локализованными датами, отметками времени, числами, текстом и пользовательскими интерфейсами.
- Эффективное использование кода, написанного на других языках, с помощью платформенно-ориентированных методов из прикладного интерфейса JNI.

Подробное рассмотрение основных языковых средств Java, включая объекты, классы, наследование, интерфейсы, события, исключения, графику, основные компоненты библиотек Swing и AWT, обобщения, многопоточную обработку и отладку программ, предлагается в первом томе настоящего издания.

КЕЙ ХОРСТМАНН — профессор факультета вычислительной техники в Университете Сан-Хосе. Имеет звание "Чемпион по Java" и является частым докладчиком на многих отраслевых конференциях. Автор обоих томов настоящего издания, а также книг *Scala for Impatient, Second Edition* (издательство Addison-Wesley, 1217 г.), *Core Java 9 for the Impatient, Second Edition* (Addison-Wesley, 2018 г.; в русском переводе книга вышла под названием *Java SE 9. Базовый курс* в издательстве "Диалектика", 2018 г.). В целом он написал более десятка книг для профессиональных программистов и студентов, изучающих вычислительную технику.

Категория: программирование
Предмет рассмотрения: язык Java, версии 9–11
Уровень: промежуточный/продвинутый

www.informit.com/java
horstmann.com

ISBN 978-5-907144-38-5

ДИАЛЕКТИКА
www.dialektika.com

Pearson