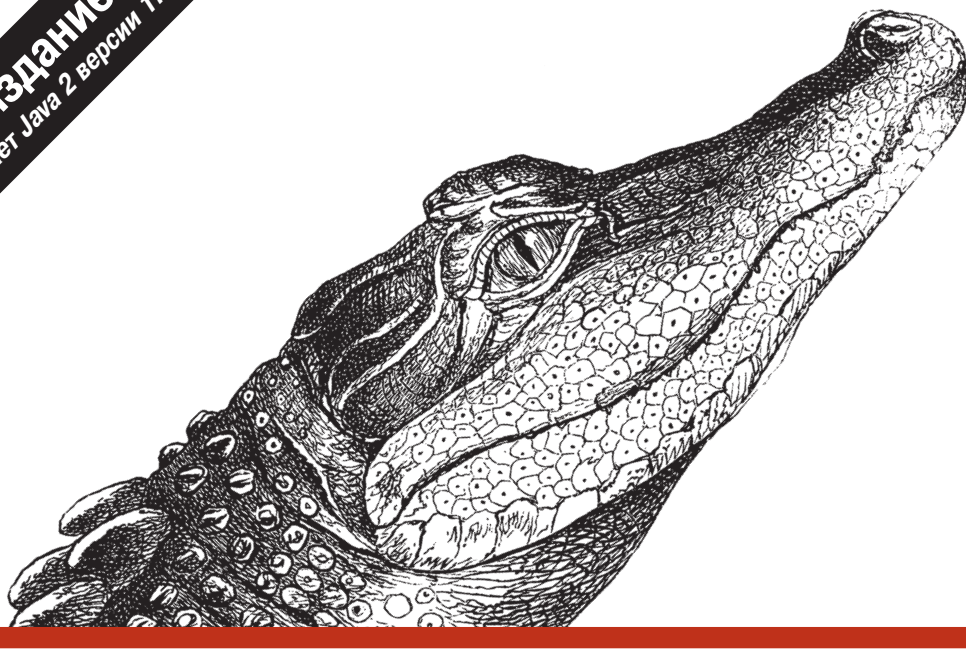
2-е издание
Охватывает Java 2 версии 1.3



JAVA™ В ПРИМЕРАХ

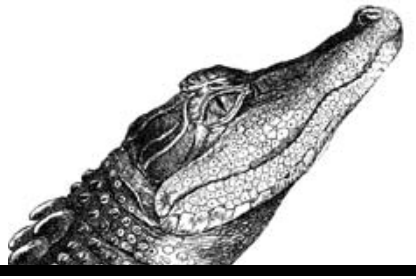
СПРАВОЧНИК

Учебное пособие к книге «Java. Справочник»

СИМВОЛ®

O'REILLY®

Дэвид Флэнаган



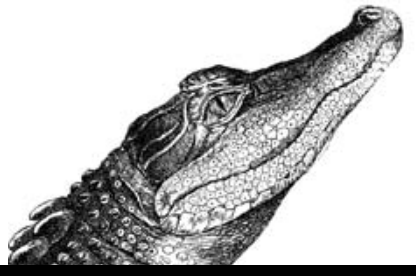
JAVA EXAMPLES IN A NUTSHELL

A Tutorial Companion to Java in a Nutshell

Second Edition

David Flanagan

O'REILLY®



JAVA В ПРИМЕРАХ

СПРАВОЧНИК

Учебное пособие к книге «Java. Справочник»

Второе издание

Дэвид Флэнаган

Санкт-Петербург — Москва
2003

Дэвид Флэнаган

Java в примерах. Справочник, 2-е издание

Перевод И. Асеева и И. Васильева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научные редакторы	*И. Васильев* *В. Шальнев*
Редактор	*В. Кузнецов*
Корректор	*С. Беляева*
Верстка	*Н. Гриценко*

Флэнаган Д.

Java в примерах. Справочник, 2-е издание – Пер. с англ. – СПб: Символ-Плюс, 2003. – 664 с., ил.

ISBN 5-93286-042-1

Второе издание книги «Java в примерах. Справочник» содержит 164 законченных практических примера: свыше 17 900 строк тщательно прокомментированного, профессионально написанного Java-кода, работающего с 20 различными программными интерфейсами Java, такими как сервлеты, JSP, XML, Swing и Java 2D. Приведены примеры, иллюстрирующие ключевые интерфейсы Java для корпоративных проектов, включая вызов удаленных методов (RMI), доступ к базам данных (JDBC). Автор бестселлера «Java in a Nutshell» (в русском переводе «Java. Справочник», Символ-Плюс) создал целую книгу примеров программ, на которых можно учиться и которые можно модифицировать для использования в своих приложениях. Если вы предпочитаете учиться «на примерах», эта книга для вас.

Книга дополняет серию справочников по Java издательства O'Reilly и будет полезна как начинающим, так и опытным Java-программистам. Удобный указатель примеров (глава 20) позволяет быстро найти метод или класс Java, а затем отыскать примеры, демонстрирующие их применение.

ISBN 5-93286-042-1
ISBN 0-596-00039-1 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 13.10.2003. Формат 70x100 1/16 . Печать офсетная.
Объем 41,5 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Эта книга из той же серии, что и мои предыдущие книги «Java in a Nutshell»[1], «Java Foundation Classes in a Nutshell» и «Java Enterprise in a Nutshell». Хотя эти книги являются, по существу, справочниками, они также содержат краткие введения в различные темы, относящиеся к программированию на Java™, и небольшие наборы примеров программ. Я писал книгу «Java в примерах», чтобы продолжить начатые в тех книгах темы, предоставив подборку примеров программ, полезных как для начинающих, так и для опытных программистов на Java.

Писать эту книгу было очень забавно. Первое издание по времени почти совпало с выходом Java версии 1.1, которая по размеру превосходила версию Java 1.0 более чем в 2 раза. Пока я был занят написанием дополнительных примеров для второго издания «Java in a Nutshell», разработчики корпорации Sun были заняты превращением Java в нечто такое, что больше не могло уложиться в рамки справочника. Из-за этого раздел, содержащий краткий справочник, так разросся, что книга «Java in a Nutshell» уже не могла вместить много примеров. Мы могли бы включить некоторые примеры по новым возможностям Java 1.1, но нам пришлось бы урезать гораздо больше, чем мы могли вставить. Это было трудное решение; примеры в «Java in a Nutshell» были одной из самых популярных ее составляющих.

Эта книга – результат тех сокращений, и я весьма доволен решением, которое мы тогда приняли. Получив свободу посвятить примерам всю книгу, я смог написать те примеры, которые я действительно хотел написать. Я смог погрузиться в тему так глубоко, как никогда раньше, и я по-настоящему наслаждался исследованием и экспериментами,

[1] Дэвид Флэнаган «Java. Справочник», 4-е издание, Символ-Плюс, 2003.

которые проводил при создании примеров. Для второго издания книги я имел удовольствие исследовать и экспериментировать с новыми разделами Java API: Swing™, Java 2D™, сервлетами и XML. Я надеюсь, что вы будете использовать эти примеры как отправные точки для ваших собственных исследований и ощутите вкус такого же волнения, какое я испытывал при их написании.

Как и сказано в ее названии, обучение в этой книге строится на примерах, на которых большинство людей обучается быстрее всего. Она не ведет вас за руку и не содержит подробного описания точного синтаксиса и операторов Java. Эта книга предназначена для совместной работы с книгами «Java in a Nutshell», «Java Foundation Classes in a Nutshell» и «Java Enterprise in a Nutshell». Изучая примеры из нее, вы наверняка оцените полезность этих изданий. Вас могут также заинтересовать и другие книги издательства O'Reilly из серий, посвященных языку Java. Список этих книг находится на сайте *http://java.oreilly.com*.

Эта книга состоит из трех частей. В главах с 1 по 9 рассматриваются основные неграфические разделы Java API. API, упомянутые в этих главах, документированы в книге «Java in a Nutshell». Главы с 10 по 15, составляющие вторую часть книги, демонстрируют графику Java и API графического пользовательского интерфейса, которые подробно рассмотрены в «Java Foundation Classes in a Nutshell». Наконец, главы с 16 по 19 содержат примеры Java API для корпоративных проектов и дополняют книгу «Java Enterprise in a Nutshell».

Вы можете читать главы этой книги в более или менее произвольном порядке, в каком они заинтересуют вас. Однако между главами имеются некоторые взаимозависимости, и некоторые из глав действительно нужно читать в том порядке, в котором они представлены. Например, прежде чем переходить к главе 5 «Сетевые операции», важно изучить главу 3 «Ввод/вывод». Глава 1 «Основы Java» и глава 2 «Объекты, классы и интерфейсы» предназначены для программистов, только начинающих работать с Java. Опытные Java-программисты, скорее всего, захотят пропустить их.

Примеры Java online

Примеры этой книги доступны в Интернете, поэтому вам не нужно набирать их все вручную! Вы можете загрузить их с веб-сайта автора *http://www.davidflanagan.com/javaexamples2* или с сайта издателя *http://www.oreilly.com/catalog/jenut2*.[1] На сайте издателя вы также можете найти список обнаруженных ошибок и опечаток. Примеры являются бесплатными для некоммерческого использования. Однако при желании использовать их в коммерческих целях необходимо оп-

[1] Речь здесь и ниже идет об оригинальном издании от O'Reilly. – *Примеч. ред.*

латить номинальную стоимость коммерческой лицензии. За подробной информацией по лицензированию обращайтесь на сайт *http://www.davidflanagan.com/javaexamples2*.

Другие книги от O'Reilly

Издательство O'Reilly издает все серии книг по Java. В их число входят «Java in a Nutshell», «Java Foundation Classes in a Nutshell» и «Java Enterprise in a Nutshell», которые, как сказано выше, являются спутниками этой книги.

С данной книгой связан справочник «Java Power Reference». Это электронный справочник по Java на компакт-диске, выполненный в стиле «Java in a Nutshell». Но будучи разработанным для просмотра в броузере, он полностью оснащен гиперссылками и включает мощный механизм поиска. Он шире по охвату, но менее глубок, чем книги серии «Java in a Nutshell». Справочник «Java Power Reference» охватывает все API платформы Java 2 и, кроме того, API многих стандартных расширений. Но он не предоставляет учебных разделов по различным API и не содержит описаний отдельных классов.

Вы можете найти полный список книг по Java от издательства O'Reilly на сайте *http://java.oreilly.com*. Отдельные главы этой книги ссылаются на специализированные книги, которые помогут вам изучить этот материал более подробно.

Соглашения, используемые в этой книге

В этой книге используются следующие соглашения по шрифтовому оформлению:

Курсивное начертание

Применяется для выделения и указания первого вхождения термина. Курсивом также выделяются команды, адреса электронной почты, веб-сайты, FTP-сайты, имена файлов и каталогов, группы новостей.

Полужирное начертание

Иногда используется для выделения клавиш клавиатуры или элементов пользовательского интерфейса, таких как кнопка **Back** или меню **Options**.

`Моноширинный шрифт`

Используется во всем Java-коде и вообще для всего, что вы набираете на клавиатуре при программировании, включая ключевые слова, типы данных, константы, имена методов, переменные, имена классов и интерфейсов. Кроме того, используется для командных строк и параметров, которые должны печататься на экране дословно, а также для тегов, которые могут появляться в HTML-документе.

Наклонный моноширинный шрифт

Используется для выделения имен параметров методов, а во многих случаях – в качестве метки-заполнителя, указывающей элемент, который в вашей программе должен быть заменен фактическим значением. Также используется для переменных выражений в параметрах командной строки.

Присылайте комментарии

Информация в этой книге была просмотрена и проверена, но вы можете обнаружить, что некоторые средства изменились (или даже найти ошибки!). Вы можете послать сообщение о любых найденных вами ошибках, а также предложения по будущим изданиям по адресу:

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
(800) 998-9938 (для США и Канады)
(707) 829-0515 (международный/местный)
(707) 829-0104 (факс)

Вы можете посылать и электронные сообщения. Для внесения вас в список рассылки или запроса каталога отправьте электронное письмо по адресу:

info@oreilly.com

Чтобы задать технический вопрос или отправить комментарий по данной книге, отправьте электронное письмо по адресу:

bookquestions@oreilly.com

На веб-сайте этой книги находятся список примеров, опечаток и планы на будущие издания. Прежде чем отправить сообщение об ошибке, проверьте список опечаток, чтобы выяснить, нет ли там уже сообщения о ней. К этой странице вы можете обратиться по адресу:

http://www.oreilly.com/catalog/jenut2

Дополнительную информацию по этой и другим книгам можно получить на веб-сайте компании O'Reilly:

http://www.oreilly.com

Благодарности

Мои благодарности, как всегда, редактору Пауле Фергюсон (Paula Ferguson), собравшей все в одну последовательную книгу и терпимой к моим многократным нарушениям графика. Благодарю также Франка Виллисона (Frank Willison) и Тима О'Рейли (Tim O'Reilly) за их желание и энтузиазм взяться за книгу, состоящую только из примеров.

В работе над этой книгой мне помогали авторы других книг по Java из издательства O'Reilly. Джонатан Нудсен (Jonathan Knudsen), автор нескольких книг по Java от O'Reilly, просмотрел главы по графике и печати. Боб Экштейн (Bob Eckstein), соавтор книги «Java Swing», просмотрел главу по Swing. Джейсон Хантер (Jason Hunter), автор книги «Java Servlet Programming», просмотрел главу по сервлетам. Ханс Бергстен (Hans Bergsten), автор выходящей книги по JavaServer Pages™, также просмотрел главу по сервлетам, но особенно тщательно изучил примеры по JSP. Бретт Мак-Лахлин (Brett McLaughlin), автор книги «Java and XML»[1], просмотрел главу по XML. Джордж Риз (George Reese), автор книги «Database programming with JDBC and Java», любезно согласился просмотреть главу по базам данных. Джим Фарли (Jim Farley), автор книги «Java Distributed Computing» и соавтор книги «Java Enterprise in a Nutshell», просмотрел примеры по RMI. Мастерство, приложенное этими рецензентами, значительно улучшило качество моих примеров. Я обязан им всем и настоятельно рекомендую их книги!

Группа подготовки издания в O'Reilly&Associates в очередной раз выполнила грандиозную работу по превращению предоставленной мною рукописи в превосходную книгу. И, как обычно, я благодарен и восхищаюсь ими.

В заключение хочу поблагодарить Кристи (Christie) – по причинам, которых слишком много, чтобы приводить их здесь.

Дэвид Флэнаган (David Flanagan)
http://www.davidflanagan.com
Июль 2000

[1] Брет Мак-Лахлин «Java и XML», 2-е издание, Символ-Плюс, 2002.

Глава 8

Отражение

API Reflection (Отражение) позволяет Java-программам инспектировать самое себя и манипулировать собой; он включает в себя класс `java.lang.Class` и пакет `java.lang.reflect`, представляющие члены некоторого класса при помощи объектов `Method`, `Constructor` и `Field`.

Reflection может получать информацию о классе и его членах. Это технология, которой пользуется, например, механизм интроспекции JavaBeans (см. главу 14 «JavaBeans») для определения свойств, событий и методов, поддерживаемых компонентом. Reflection в Java может также манипулировать объектами. Класс `Field` можно использовать как для опроса содержимого полей объекта, так и для установки их значений, класс `Method` – для вызова методов, а класс `Constructor` – для создания новых объектов. Примеры в этой главе демонстрируют как инспектирование объектов, так и манипулирование ими при помощи API Reflection.

Помимо примеров этой главы API Reflection используется также в примере из главы 17 «Доступ к базам данных при помощи SQL».

Получение информации о классах и членах

Пример 8.1 показывает программу, использующую классы `Class`, `Constructor`, `Field` и `Method` для отображения информации о заданном классе. Вывод этой программы подобен кратким описаниям классов, приведенным в книге «Java in a Nutshell». (Вы можете заметить, что имена аргументов методов не показаны; имена аргументов не сохраняются в файле класса, поэтому они не могут быть получены через API Reflection.)

Здесь представлен вывод, полученный в результате применения ShowClass к нему самому:

```
public class ShowClass extends Object {
 // Constructors
 public ShowClass();
 // Fields
 // Methods
 public static void main(String[]) throws ClassNotFoundException;
 public static String modifiers(int);
 public static void print_class(Class);
 public static String typename(Class);
 public static void print_field(Field);
 public static void print_method_or_constructor(Member);
}
```

Код этого примера совершенно прозрачен. Здесь используется метод Class.forName() для динамической загрузки класса, заданного по имени, а затем вызываются различные методы объекта Class для просмотра родительского класса, интерфейсов и членов класса. В примере используются объекты Constructor, Field и Method для получения информации о каждом члене класса.

Пример 8.1. ShowClass.java

```
package com.davidflanagan.examples.reflect;
import java.lang.reflect.*;

/** Программа, выдающая краткое описание класса, заданного по имени */
public class ShowClass {
  /** Метод main(). Печатает информацию о заданном классе */
  public static void main(String[] args) throws ClassNotFoundException {
    Class c = Class.forName(args[0]);
    print_class(c);
  }

  /**
   * Отображаем модификаторы, имя, родительский класс и интерфейсы класса
   * или интерфейса. Затем выводим список его конструкторов, полей и методов.
   **/
  public static void print_class(Class c)
  {
    // Печатаем модификаторы, тип (класс или интерфейс), имя,
    // родительский класс.
    if (c.isInterface()) {
      // Модификаторы будут здесь включать в себя ключевое слово «interface»...
      System.out.print(Modifier.toString(c.getModifiers()) + " " +
                       typename(c));
    }
    else if (c.getSuperclass() != null) {
      System.out.print(Modifier.toString(c.getModifiers()) + " class " +
                       typename(c) +
```

```
            " extends " + typename(c.getSuperclass()));
    }
    else {
      System.out.print(Modifier.toString(c.getModifiers()) + " class " +
            typename(c));
    }

    // Печатаем интерфейсы или суперинтерфейсы класса или интерфейса.
    Class[] interfaces = c.getInterfaces();
    if ((interfaces != null) && (interfaces.length > 0)) {
      if (c.isInterface()) System.out.print(" extends ");
      else System.out.print(" implements ");
      for(int i = 0; i < interfaces.length; i++) {
        if (i > 0) System.out.print(", ");
        System.out.print(typename(interfaces[i]));
      }
    }

    System.out.println(" {");      // Начало списка членов класса.

    // Теперь просматриваем и печатаем члены класса.
    System.out.println(" // Constructors");
    Constructor[] constructors = c.getDeclaredConstructors();
    for(int i = 0; i < constructors.length; i++)  // Выводим конструкторы.
      print_method_or_constructor(constructors[i]);

    System.out.println(" // Fields");
    Field[] fields = c.getDeclaredFields();       // Просматриваем поля.
    for(int i = 0; i < fields.length; i++)        // Отображаем их.
      print_field(fields[i]);

    System.out.println(" // Methods");
    Method[] methods = c.getDeclaredMethods();     // Просматриваем методы.
    for(int i = 0; i < methods.length; i++)        // Отображаем их.
      print_method_or_constructor(methods[i]);

    System.out.println("}");       // Конец списка членов класса.
  }

  /** Возвращаем имя интерфейса или примитивного типа, обрабатывая массивы. */
  public static String typename(Class t) {
    String brackets = "";
    while(t.isArray()) {
      brackets += "[]";
      t = t.getComponentType();
    }
    String name = t.getName();
    int pos = name.lastIndexOf('.');
    if (pos != -1) name = name.substring(pos+1);
    return name + brackets;
  }

  /** Возвращаем строковую версию модификаторов,
      для красоты применяя пробелы. */
```

```
  public static String modifiers(int m) {
    if (m == 0) return "";
    else return Modifier.toString(m) + " ";
  }

  /** Печатаем модификаторы, тип и имя поля. */
  public static void print_field(Field f) {
    System.out.println(" " + modifiers(f.getModifiers()) +
               typename(f.getType()) + " " + f.getName() + ";");
  }

  /**
   * Печатаем модификаторы, возвращаем тип, имя, типы параметров и типы
   * исключений метода или конструктора. Обратите внимание на применение
   * интерфейса Member, позволяющее этому методу работать как
   * с объектами Method, так и с объектами Constructor
   **/
  public static void print_method_or_constructor(Member member) {
    Class returntype=null, parameters[], exceptions[];
    if (member instanceof Method) {
      Method m = (Method) member;
      returntype = m.getReturnType();
      parameters = m.getParameterTypes();
      exceptions = m.getExceptionTypes();
      System.out.print(" " + modifiers(member.getModifiers()) +
                typename(returntype) + " " + member.getName() +
                "(");
    } else {
      Constructor c = (Constructor) member;
      parameters = c.getParameterTypes();
      exceptions = c.getExceptionTypes();
      System.out.print(" " + modifiers(member.getModifiers()) +
                typename(c.getDeclaringClass()) + "(");
    }

    for(int i = 0; i < parameters.length; i++) {
      if (i > 0) System.out.print(", ");
      System.out.print(typename(parameters[i]));
    }
    System.out.print(")");
    if (exceptions.length > 0) System.out.print(" throws ");
    for(int i = 0; i < exceptions.length; i++) {
      if (i > 0) System.out.print(", ");
      System.out.print(typename(exceptions[i]));
    }
    System.out.println(";");
  }
}
```

Вызов метода, заданного по имени

Пример 8.2 определяет класс Command, демонстрирующий еще одно использование Reflection API. Объект Command включает в себя объект Method, объект, метод которого вызывается, и массив переменных, передаваемых этому методу. Метод invoke() вызывает метод заданного объекта, используя заданные аргументы. Метод actionPerformed() делает то же самое. Если вы уже прочитали главу 10 «Графические интерфейсы пользователя (GUI)», вы знаете, что этот метод реализует интерфейс java.awt.event.ActionListener, из чего следует, что объекты Command могут использоваться как слушатели действий, позволяя реагировать на нажатие кнопок, выбор пунктов меню и другие события, относящиеся к графическому интерфейсу пользователя. Для того чтобы обработать события, GUI-программы обычно создают набор реализаций интерфейса ActionListener. При использовании класса Command слушатели действий могут определяться без необходимости создания множества новых классов.

Самой полезной вещью (при этом с самым сложным кодом) в классе Command является метод parse(), разбирающий строку, которая содержит имя метода и список аргументов, для создания объекта Command. Он полезен, например, потому что позволяет считывать объекты Command из конфигурационных файлов. Мы воспользуемся этой возможностью класса Command в главе 10.

Java не позволяет передавать методы прямо, подобно данным, но Reflection API делает возможным косвенный вызов методов, заданных их именами. Заметим, что этот прием не особенно эффективен. Но для асинхронной обработки событий в GUI он достаточно эффективен: косвенные вызовы методов через Reflection API действуют гораздо быстрее, чем требуется для удовлетворения ограниченных потребностей человеческого восприятия. Однако вызов методов по имени оказывается неприемлемым, когда необходимы повторные вызовы или когда компьютер не ожидает ввода от человека. Таким образом, этот прием не следует использовать, например, для передачи метода сравнения в процедуру сортировки или для передачи фильтра имен файлов методу, выдающему листинг каталога. В подобных случаях следует применять стандартную технику реализации класса, содержащего нужный метод, и передачи экземпляра класса соответствующей процедуре.

Пример 8.2. Command.java

```
package com.davidflanagan.examples.reflect;
import java.awt.event.*;
import java.beans.*;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;
```

```
/**
 * Этот класс представляет объект Method, список аргументов, передаваемых
 * этому методу, и объект, из которого этот метод должен вызываться.
 * Метод invoke() вызывает метод. Метод actionPerformed() делает то же
 * самое, позволяя этому классу реализовывать ActionListener и использоваться
 * для реакции на события ActionEvents, генерируемые в GUI или где-либо еще.
 * Статический метод parse() разбирает строку, представляющую метод,
 * и его аргументы.
 **/
public class Command implements ActionListener {
  Method m;      // Вызываемый метод
  Object target; // Объект, из которого вызывается метод
  Object[] args; // Аргументы, передаваемые методу

  // Пустой массив; используется для методов, не имеющих аргументов.
  static final Object[] nullargs = new Object[] {};

  /** Этот конструктор создает объект Command для метода без аргументов */
  public Command(Object target, Method m) { this(target, m, nullargs); }

  /**
   * Этот конструктор создает объект Command для метода, принимающего
   * заданный массив аргументов. Отметим, что метод parse() предоставляет
   * еще один способ создания объекта Command.
   **/
  public Command(Object target, Method m, Object[] args) {
    this.target = target;
    this.m = m;
    this.args = args;
  }

  /**
   * Вызывает Command посредством вызова метода из его объекта target
   * и передачи аргументов. См. также actionPerformed(), который,
   * в отличие от этого метода, не выдает проверяемых исключений.
   **/
  public void invoke()
    throws IllegalAccessException, InvocationTargetException
  {
    m.invoke(target, args); // Используем отражение для вызова метода
  }

  /**
   * Этот метод реализует интерфейс ActionListener. Он подобен методу invoke()
   * с тем отличием, что перехватывает выданные этим методом исключения
   * и передает их в виде непроверяемого исключения RuntimeException
   **/
  public void actionPerformed(ActionEvent e) {
    try {
      invoke();              // Обращаемся к вызывающему методу,
    }
    catch (InvocationTargetException ex) { // но обрабатываем исключения
      throw new RuntimeException("Command: " +
```

```
                    ex.getTargetException().toString());
    }
    catch (IllegalAccessException ex) {
      throw new RuntimeException("Command: " + ex.toString());
    }
  }

  /**
   * Этот статический метод создает объект Command с использованием заданного
   * объекта target и заданной строки. Эта строка должна содержать имя
   * метода, за которым следуют необязательный, заключенный в скобки список
   * аргументов, разделенных запятыми, и точка с запятой. Аргументы могут
   * быть литералами логическими, целыми или типа double, или заключенными
   * в двойные кавычки строками. Анализатор терпимо относится к пропущенным
   * запятым, точкам с запятой и кавычкам, но выдает IOException,
   * если ему не удается разобрать строку.
   **/
  public static Command parse(Object target, String text) throws IOException
  {
    String methodname;                  // Имя метода
    ArrayList args = new ArrayList();   // Будет содержать
                                        // разобранные аргументы.
    ArrayList types = new ArrayList(); // Будет содержать типы аргументов.

    // Преобразуем строку в символьный поток и применяем класс
    // StreamTokenizer для преобразование его в поток лексем
    // (маркеров, tokens)
    StreamTokenizer t = new StreamTokenizer(new StringReader(text));

    // Первой лексемой должно быть имя метода
    int c = t.nextToken();    // читаем лексему
    if (c != t.TT_WORD)       // проверяем ее тип
      throw new IOException("Отсутствует имя метода для команды");
    methodname = t.sval;      // Запоминаем имя метода

    // Далее должна следовать либо точка с запятой, либо открывающая скобка
    c = t.nextToken();
    if (c == '(') { // Увидев открывающую скобку, разбираем список аргументов
      for(;;) {               // Цикл до конца списка аргументов
        c = t.nextToken();    // Читаем следующую лексему

        if (c == ')') {       // Проверяем, не конец ли это списка.
          c = t.nextToken(); // Если да, выделяем необязательную
                             // точку с запятой
          if (c != ';') t.pushBack();
          break;              // Выходим из цикла.
        }

        // Если нет, лексема является аргументом; определяем ее тип
        if (c == t.TT_WORD) {
          // Если лексема является идентификатором (identifier), выделяем
          // логические литералы, а все другие лексемы интерпретируем
          // как строковые литералы без кавычек.
          if (t.sval.equals("true")) {    // Логический литерал
```

```
          args.add(Boolean.TRUE);
          types.add(boolean.class);
        }
        else if (t.sval.equals("false")) { // Логический литерал
          args.add(Boolean.FALSE);
          types.add(boolean.class);
        }
        else {               // Считаем, что это строка
          args.add(t.sval);
          types.add(String.class);
        }
      }
      else if (c == '"') {     // Если лексема - это строка в кавычках
        args.add(t.sval);
        types.add(String.class);
      }
      else if (c == t.TT_NUMBER) { // Если лексема - это число
        int i = (int) t.nval;
        if (i == t.nval) {      // Проверяем, целое ли оно
          // Замечание: этот код проинтерпретирует лексему
          // вида "2.0" как целое число!
          args.add(new Integer(i));
          types.add(int.class);
        }
        else {      // В противном случае оно принадлежит типу double
          args.add(new Double(t.nval));
          types.add(double.class);
        }
      }
      else {            // Любая другая лексема является ошибкой
        throw new IOException("Неизвестная лексема " + t.sval +
                    " в списке аргументов метода " +
                    methodname + "().");
      }

      // Далее должна следовать запятая, но если это окажется не так,
      // мы жаловаться не будем
      c = t.nextToken();
      if (c != ',') t.pushBack();
    }
  }
  else if (c != ';') { // Если за именем метода не следует запятая,
    t.pushBack();      // ожидаем точку с запятой, но не настаиваем на этом.
  }

  // Мы разобрали список аргументов.
  // Теперь преобразуем списки аргументов и их типов в массивы
  Object[] argValues = args.toArray();
  Class[] argtypes = (Class[])types.toArray(new Class[argValues.length]);

  // В этом месте у нас уже есть имя метода и массивы типов и значений
  // аргументов. Применяем отражение к классу объекта target,
```

```
    // чтобы найти метод с заданными именем и типами аргументов. Если
    // метода с заданным именем не нашлось, выдаем исключение.
    Method method;
    try { method = target.getClass().getMethod(methodname, argtypes); }
    catch (Exception e) {
      throw new IOException("Нет такого метода или неправильно заданы " +
                  "типы аргументов: " + methodname);
    }

    // Наконец, создаем и возвращаем объект Command, используя переданный
    // этому методу объект target, полученный выше объект Method и
    // массив аргументов, выделенный из строки.
    return new Command(target, method, argValues);
  }

  /**
   * Эта простая программа показывает, как объект Command может быть выделен
   * из строки и использован в качестве объекта ActionListener
   * в Swing-приложении.
   **/
  static class Test {
    public static void main(String[] args) throws IOException {
      javax.swing.JFrame f = new javax.swing.JFrame("Command Test");
      javax.swing.JButton b1 = new javax.swing.JButton("Tick");
      javax.swing.JButton b2 = new javax.swing.JButton("Tock");
      javax.swing.JLabel label = new javax.swing.JLabel("Hello world");
      java.awt.Container pane = f.getContentPane();

      pane.add(b1, java.awt.BorderLayout.WEST);
      pane.add(b2, java.awt.BorderLayout.EAST);
      pane.add(label, java.awt.BorderLayout.NORTH);

      b1.addActionListener(Command.parse(label, "setText(\"tick\");"));
      b2.addActionListener(Command.parse(label, "setText(\"tock\");"));

      f.pack();
      f.show();
    }
  }
}
```

Упражнения

8-1. Напишите программу, получающую в качестве заданного в командной строке аргумента класс Java и использующую класс `Class` для печати всех родительских классов этого класса. Вызванная, например, с аргументом «java.awt.Applet», эта программа должна напечатать следующее: `java.lang.Object java.awt.Component java.awt.Container java.awt.Panel`.

8-2. Переделайте программу, написанную вами в упражнении 8-1, чтобы она распечатывала все интерфейсы, реализованные задан-

ным классом или любым из его родительских классов. Обработайте случай, когда классы реализуют интерфейсы, дочерние по отношению к другим интерфейсам. Если класс реализует, например, `java.awt.LayoutManager2`, а интерфейсы `LayoutManager2` и `LayoutManager` являются родительскими по отношению к первому, они тоже должны быть включены в список.

8-3. На основе класса `Command` из примера 8.2 определите класс `Assignment`. Вместо вызова заданного по имени метода, как это делает `Command`, `Assignment` должен присваивать значение заданному по имени полю объекта. Пусть у вашего класса будет конструктор со следующей сигнатурой:

```
public Assignment(Object target, Field field, Object value)
```

Включите в него также метод `assign()`, который, будучи вызван, присвоит заданное значение заданному полю заданного объекта. Класс `Assignment` должен реализовать `ActionListener` и определять метод `actionPerformed()`, также выполняющий присваивание. Напишите для `Assignment` статический метод `parse()`, подобный методу `parse()` для `Command`. Он должен разбирать строки вида *`fieldname=value`* и уметь выделять логические, числовые и строковые значения.

Глава 9

Сериализация объектов

Сериализация объектов состоит в способности класса `Serializable` выводить состояние экземпляра объекта в байтовый поток, а позже снова считывать это состояние, создавая копию исходного объекта. При сериализации объекта вместе с ним подвергается сериализации вся иерархия объектов, на которые ссылается сериализуемый объект. Это обеспечивает возможность сериализации сложных структур данных, таких как двоичные деревья. Также можно сериализовать апплеты и полные иерархии компонентов GUI.

Простая сериализация

Несмотря на мощь и важность сериализации, она осуществляется с использованием простого API, входящего в пакет `java.io`: объект сериализуется при помощи метода `writeObject()` класса `ObjectOutputStream` и десериализуется методом `readObject()` класса `ObjectInputStream`. Эти классы являются байтовыми потоками, подобными различным другим потокам, какие мы видели в главе 3 «Ввод/вывод». Они реализуют интерфейсы `ObjectOutput` и `ObjectInput` соответственно, а эти интерфейсы являются в свою очередь дочерними для интерфейсов `DataOutput` и `DataInput`. Это значит, что `ObjectOutputStream` определяет те же методы, что и `DataOutputStream` для записи примитивных значений, тогда как `ObjectInputStream` определяет те же методы, что и `DataInputStream`, для чтения примитивных значений. Однако здесь нас интересуют методы `writeObject()` и `readObject()`, которые записывают и считывают объекты.

Сериализовать можно только объекты, реализующие интерфейс `java.io.Serializable`. Интерфейс `Serializable` — это пустой интерфейс; он не определяет никаких методов, подлежащих реализации. Тем не менее

для некоторых классов раскрытие их состояния при использовании механизма сериализации оказывается нежелательным по соображениям безопасности. Поэтому, чтобы класс реализовал этот интерфейс, он должен явно объявить себя в качестве сериализуемого.

Объект сериализуется путем передачи его методу `writeObject()` класса `ObjectOutputStream`. Он выводит значения всех полей объекта, включая закрытые (private) поля и поля, унаследованные от родительских классов. Значения примитивных полей просто записываются в поток, как это делалось бы с `DataOutputStream`. Однако когда поле объекта ссылается на другой объект, массив или строку, метод `writeObject()` рекурсивно вызывается для сериализации также и этого объекта. Если этот объект (или элемент массива) ссылается еще на один объект, `writeObject()` снова рекурсивно вызывается. Таким образом, один вызов `writeObject()` может в результате привести к сериализации всей иерархии объектов. Когда два или большее число объектов ссылаются друг на друга, алгоритм сериализации обеспечивает лишь по одному выводу для каждого объекта; `writeObject()` не может войти в бесконечную рекурсию.

Десериализация объекта просто проходит этот процесс в обратном направлении. Объект считывается из потока данных посредством вызова метода `readObject()` класса `ObjectInputStream`. Он восстанавливает объект в том состоянии, в котором он находился при сериализации. Если объект ссылается на другие объекты, они также рекурсивно десериализуются.

Пример 9.1 демонстрирует основы сериализации. Этот пример определяет общие методы, способные сохранять в файле и получать из него состояние любого сериализуемого объекта. Также он включает в себя интересный метод `deepclone()`, использующий сериализацию для копирования иерархии объектов. В пример входят внутренний класс `Serializable` и тестовый класс, демонстрирующий эти методы в работе.

Пример 9.1. Serializer.java

```
package com.davidflanagan.examples.serialization;
import java.io.*;

/**
 * Этот класс определяет утилиты, использующие Java-сериализацию.
 **/
public class Serializer {
    /**
     * Сериализуем объект o (и все объекты, на которые он ссылается
     * и объявленные как Serializable) и его состояние сохраняем в файле f.
     **/
    static void store(Serializable o, File f) throws IOException {
        ObjectOutputStream out =  // Класс для сериализации
            new ObjectOutputStream(new FileOutputStream(f));
        out.writeObject(o);       // Этот метод сериализует иерархию объектов
        out.close();
    }
```

```
/**
 * Десериализируем содержимое файла f и возвращаем результирующий объект
 **/
static Object load(File f) throws IOException, ClassNotFoundException {
    ObjectInputStream in =    // Класс для десериализации
        new ObjectInputStream(new FileInputStream(f));
    return in.readObject();   // Этот метод десериализует иерархию объектов
}

/**
 * Используем сериализацию объектов для создания полного клона
 * («deep clone») объекта o. Этот метод сериализует объект o и все
 * объекты,на которые тот ссылается, а затем десериализует эту иерархию
 * объектов, т. е. все копируется. Метод deepclone() отличается от метода
 * clone(), который обычно реализуется так, чтобы создавался частичный
 * клон («shallow» clone), копирующий ссылки на другие объекты, а не сами
 * объекты, на которые направлены ссылки
 **/
static Object deepclone(final Serializable o)
    throws IOException, ClassNotFoundException
{
    // Создаем пару канальных (pipe) потоков.
    // Будем записывать байты в один из них, а затем читать из другого.
    final PipedOutputStream pipeout = new PipedOutputStream();
    PipedInputStream pipein = new PipedInputStream(pipeout);

    // Теперь создаем отдельный поток исполнения для сериализации объекта
    // и записи его байтов в PipedOutputStream
    Thread writer = new Thread() {
            public void run() {
                ObjectOutputStream out = null;
                try {
                    out = new ObjectOutputStream(pipeout);
                    out.writeObject(o);
                }
                catch(IOException e) {}
                finally {
                    try { out.close(); } catch (Exception e) {}
                }
            }
        };
    writer.start();  // Запускаем поток исполнения на сериализацию и запись

    // В то же время в этом потоке исполнения читаем и десериализуем
    // данные из канального потока ввода. Получившийся в результате
    // объект будет полным клоном оригинала.
    ObjectInputStream in = new ObjectInputStream(pipein);
    return in.readObject();
}

/**
 * Это простая сериализуемая структура данных, используемая ниже
 * для тестирования определенных выше методов.
 **/
```

```
    public static class DataStructure implements Serializable {
        String message;
        int[] data;
        DataStructure other;
        public String toString() {
            String s = message;
            for(int i = 0; i < data.length; i++)
                s += " " + data[i];
            if (other != null) s += "\n\t" + other.toString();
            return s;
        }
    }

    /** Этот класс определяет тестирующий метод main()*/
    public static class Test {
        public static void main(String[] args)
            throws IOException, ClassNotFoundException
        {
            // Создаем простую иерархию объектов
            DataStructure ds = new DataStructure();
            ds.message = "hello world";
            ds.data = new int[] { 1, 2, 3, 4 };
            ds.other = new DataStructure();
            ds.other.message = "nested structure";
            ds.other.data = new int[] { 9, 8, 7 };

            // Отображаем исходную иерархию
            System.out.println("Исходная структура данных: " + ds);

            // Выводим ее в файл
            File f = new File("datastructure.ser");
            System.out.println("Сохранение в файле...");
            Serializer.store(ds, f);

            // Считываем ее из файла обратно и снова отображаем
            ds = (DataStructure) Serializer.load(f);
            System.out.println("Считано из файла: " + ds);

            // Создаем полный клон и отображаем его. После создания копии
            // переделываем оригинал, чтобы доказать, что клон «полный».
            DataStructure ds2 = (DataStructure) Serializer.deepclone(ds);
            ds.other.message = null; ds.other.data = null; // Change original
            System.out.println("Полный клон: " + ds2);
        }
    }
}
```

Специальная сериализация

Не всякая часть состояния программы может или должна сериализоваться. Объектам `FileDescriptor`**, например, присуща зависимость от платформы или от виртуальной машины. Будучи сериализованным,** `FileDescriptor` **не имел бы, например, никакого смысла на другой вир-**

туальной машине. По этой причине, а также в силу вышеописанных важных соображений безопасности не все объекты могут быть сериализованы.

Даже когда объект является сериализуемым, полная сериализация его состояния может не иметь смысла. Некоторые поля могут быть «черновыми» – в них могут храниться временные или заранее вычисленные значения, не содержащие информации, действительно нужной при десериализации объекта. Рассмотрим компонент GUI. Он может определять поля, в которых хранятся координаты последнего полученного им щелчка мыши. Эта информация не представляет никакого интереса при десериализации компонента, поэтому нет смысла утруждать себя сохранением значений этих полей как части состояния компонента. Чтобы сообщить механизму сериализации, что поле не нужно сохранять, его просто объявляют как `transient` (временное):

```
protected transient short last_x, last_y; // Временные поля координат
                                          положения указателя мыши
```

Бывает также, что поле не является временным (другими словами, оно сохраняет важную часть состояния объекта), но по некоторым причинам оно не может быть успешно сериализовано. Рассмотрим другой компонент GUI, вычисляющий свой предпочтительный размер в зависимости от размера отображаемого им текста. Поскольку размер шрифтов несколько варьируется от платформы к платформе, заранее вычисленный предпочтительный размер перестает быть действительным, если компонент сериализован на платформе одного типа, а десериализован на платформе другого типа. Поскольку на поля с предпочтительным размером нельзя полагаться при десериализации, их следует объявлять как `transient`, чтобы они не занимали места в сериализованном объекте. Однако в этом случае их значения после десериализации объекта должны быть пересчитаны.

Класс может определять особенности поведения своих объектов при сериализации и десериализации (подобные пересчету предпочтительного размера) путем реализации методов `writeObject()` и `readObject()`. Как ни удивительно, эти методы не определены никаким интерфейсом, и их следует определять как `private`. Если класс определяет эти методы, при сериализации и десериализации объектов `ObjectOutputStream` или `ObjectInputStream` они будут вызывать соответствующий метод.

Компонент GUI мог бы, например, определить метод `readObject()`, чтобы дать ему возможность при десериализации пересчитать предпочтительные размеры. Этот метод мог бы выглядеть так:

```
private void readObject(ObjectInputStream in)
       throws IOException, ClassNotFoundException
{
 in.defaultReadObject();      // Обычная десериализация компонента.
 this.computePreferredSize(); // Но теперь его размеры пересчитываются.
}
```

Этот метод вызывает метод `defaultReadObject()` **класса** `ObjectInputStream` **для обычной десериализации объекта, а затем выполняет всю необходимую постобработку.**

Пример 9.2 – более сложный пример специальной сериализации. Он показывает класс, реализующий динамический массив целых чисел. Этот класс определяет метод `writeObject()` **для выполнения некоторой предварительной обработки перед тем, как он будет сериализован, и метод** `readObject()` **– для дополнительной обработки после десериализации.**

Пример 9.2. IntList.java

```
package com.davidflanagan.examples.serialization;
import java.io.*;

/**
 * Простой класс, реализующий динамический массив целых чисел и умеющий
 * сериализовать себя так же эффективно, как массив фиксированного размера.
 **/
public class IntList implements Serializable {
  protected int[] data = new int[8]; // Массив для хранения чисел.
  protected transient int size = 0;  // Индекс следующего неиспользованного
                                     // элемента массива

  /** Возвращается элемент массива */
  public int get(int index) throws ArrayIndexOutOfBoundsException {
    if (index >= size) throw new ArrayIndexOutOfBoundsException(index);
    else return data[index];
  }

  /** К массиву добавляется новое число; при необходимости размер
      массива увеличивается */
  public void add(int x) {
    if (data.length==size) resize(data.length*2); // При необходимости
                                                  // увеличиваем размер массива.
    data[size++] = x;                             // Сохраняем целое здесь.
  }

  /** Внутренний метод для изменения размера выделенного массиву пространства */
  protected void resize(int newsize) {
    int[] newdata = new int[newsize];           // Создаем новый массив
    System.arraycopy(data, 0, newdata, 0, size); // Копируем элементы массива.
    data = newdata;                             // Заменяем старый массив
  }

  /** Перед сериализацией массива освобождаемся от неиспользуемых
      элементов массива */
  private void writeObject(ObjectOutputStream out) throws IOException {
    if (data.length > size) resize(size); // Сжимаем массив.
    out.defaultWriteObject();       // Затем записываем его обычным образом.
  }

  /** Вычисляем временное поле size после десериализации массива */
  private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
```

```
    {
      in.defaultReadObject();      // Считываем массив обычным образом.
      size = data.length;          // Восстанавливаем значение временного поля.
    }

    /**
     * Содержит ли объект this те же значения, что и объект o? Мы замещаем
     * этот метод класса Object так, чтобы протестировать наш класс.
     **/
    public boolean equals(Object o) {
      if (!(o instanceof IntList)) return false;
      IntList that = (IntList) o;
      if (this.size != that.size) return false;
      for(int i = 0; i < this.size; i++)
        if (this.data[i] != that.data[i]) return false;
      return true;
    }

    /** Метод main() доказывает, что все это работает */
    public static void main(String[] args) throws Exception {
      IntList list = new IntList();
      for(int i = 0; i < 100; i++) list.add((int)(Math.random()*40000));
      IntList copy = (IntList)Serializer.deepclone(list);
      if (list.equals(copy)) System.out.println("equal copies");
      Serializer.store(list, new File("intlist.ser"));
    }
  }
```

Экстернализируемые классы

Интерфейс `Externalizable` **расширяет интерфейс** `Serializable` **и определяет методы** `writeExternal()` **и** `readExternal()`**. Объект** `Externalizable` **может быть сериализован, как и другие объекты** `Serializable`**, но механизм сериализации вызывает методы** `writeExternal()` **и** `readExternal()` **для выполнения сериализации и десериализации. В отличие от методов** `readObject()` **и** `writeObject()` **из примера 9.2, методы** `readExternal()` **и** `writeExternal()` **не могут вызывать методы** `defaultReadObject()` **и** `defaultWriteObject()` **— они должны считывать и записывать все состояние объекта самостоятельно.**

Объявлять объект как `Externalizable` **оказывается полезным, когда у объекта уже есть свой формат файла или когда нужно выполнить нечто, что просто невозможно сделать посредством стандартных методов сериализации. Пример 9.3 определяет класс** `CompactIntList`**, производный от интерфейса** `Externalizable` **и класса** `IntList` **из предыдущего примера. Предполагается, что класс** `CompactIntList` **будет обычно использоваться для хранения большого количества маленьких целых чисел; он реализует интерфейс** `Externalizable`**, поэтому он может определить значительно более компактную форму сериализации, чем формат, используемый классами** `ObjectOutputStream` **и** `ObjectInputStream`**.**

Пример 9.3. CompactIntList.java

```
package com.davidflanagan.examples.serialization;
import java.io.*;

/**
 * Этот подкласс IntList предполагает, что большая часть чисел, которые он
 * содержит, меньше 32 000. Чтобы использовать преимущества такой ситуации,
 * он реализует интерфейс Externalizable и определяет
 * более компактный формат сериализации.
 **/
public class CompactIntList extends IntList implements Externalizable {
  /**
   * Этот номер версии появляется здесь на тот случай, если последующим
   * версиям этого класса понадобится модифицировать формат
   * экстернализации, но вместе с тем сохранить совместимость
   * с объектами, экстернализированными по этой версии
   **/
  static final byte version = 1;

  /**
   * Этот метод из интерфейса Externalizable отвечает за полное сохранение
   * состояния объекта в заданном потоке (stream). Он может писать
   * как угодно, лишь бы readExternal() смог прочитать написанное.
   **/
  public void writeExternal(ObjectOutput out) throws IOException {
    if (data.length > size) resize(size); // Сжимаем массив.

    out.writeByte(version); // Начинаем с номера нашей версии.
    out.writeInt(size);     // Выводим количество элементов массива
    for(int i = 0; i < size; i++) { // Теперь цикл по элементам массива
      int n = data[i];              // Записываемый элемент массива
      if ((n < Short.MAX_VALUE) && (n > Short.MIN_VALUE+1)) {
        // Если значение n помещается в тип short и не равно Short.MIN_VALUE,
        // записываем его как short, экономя при этом два байта
        out.writeShort(n);
      }
      else {
        // В противном слуае сначала выводим специальное значение
        // Short.MIN_VALUE, сигнализируя этим, что число не поместилось
        // в short, а затем выводим число, используя все 4 байта.
        // Всего получается 6 байтов.
        out.writeShort(Short.MIN_VALUE);
        out.writeInt(n);
      }
    }
  }

  /**
   * Этот метод из интерфейса Externalizable отвечает за полное восстановление
   * состояния объекта. Для воссоздания объекта будет вызываться конструктор
   * без аргументов, а этот метод должен прочитать написанное
   * writeExternal(), чтобы восстановить состояние объекта.
   **/
```

```
  public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException
  {
    // Начинаем с чтения и проверки номера версии.
    byte v = in.readByte();
    if (v != version)
      throw new IOException("CompactIntList: неизвестный номер версии");

    // Считываем количество элементов массива и создаем массив
    // соответствующей величины
    int newsize = in.readInt();
    resize(newsize);
    this.size = newsize;

    // Теперь считываем из потока соответствующее количество значений
    for(int i = 0; i < newsize; i++) {
      short n = in.readShort();
      if (n != Short.MIN_VALUE) data[i] = n;
      else data[i] = in.readInt();
    }
  }

  /** Метод main()доказывает, что все это работает */
  public static void main(String[] args) throws Exception {
    CompactIntList list = new CompactIntList();
    for(int i = 0; i < 100; i++) list.add((int)(Math.random()*40000));
    CompactIntList copy = (CompactIntList)Serializer.deepclone(list);
    if (list.equals(copy)) System.out.println("equal copies");
    Serializer.store(list, new File("compactintlist.ser"));
  }
}
```

Сериализация и учет версий класса

Одной из особенностей примера 9.3 является включение номера версии в записываемый им поток (stream) сериализации. Это оказывается полезным, если класс со временем развивается и ему приходится использовать при сериализации новый формат. Номер версии позволяет будущим версиям класса распознавать сериализованные объекты, записанные этой версией класса.

Для объектов `Serializable`, не являющихся объектами `Externalizable`, интерфейс Serialization API сам поддерживает учет версий. Для сериализуемого объекта некоторая информация о классе этого объекта должна, очевидно, сериализоваться вместе с ним, чтобы при десериализации объекта мог быть загружен правильный файл класса. Эта информация о классе представляется классом `java.io.ObjectStreamClass`. Он содержит полное имя класса и номер версии для этого класса. Номер версии очень важен, поскольку более ранние версии класса, возможно, окажутся неспособными десериализовать сериализованный

экземпляр, созданный более поздней версией этого же класса. Номер версии класса имеет тип `long`. По умолчанию механизм сериализации создает уникальный номер версии посредством вычисления хеш-ключа из имени класса, имени его родительского класса и всех интерфейсов, которые он реализует, имен и типов его полей и из имен и типов его незакрытых (nonprivate) методов. Таким образом, как только в классе добавляется новый метод, изменяется имя поля или производится даже малейшее изменение API или реализации класса, его вычисляемая версия также меняется. Когда объект сериализован одной версией класса, он не может быть десериализован его версией с другим номером.

Таким образом, внесение изменений в сериализуемый класс, даже самых незначительных, не влияющих на формат сериализации, разрушает совместимость между версиями по сериализации. Например, наш класс `IntList` в действительности обязан иметь метод `set()`, устанавливающий значение заданного элемента списка. Но если добавить этот метод, новая версия класса не сможет десериализовать объекты, сериализованные старой версией. Предупредить эту проблему можно, задав версию класса явно. Это делается путем включения в класс константного поля `serialVersionUID`.

Значение этого поля не важно; оно только должно быть одним и тем же для всех версий класса, имеющих совместимый формат сериализации. Поскольку исходный класс `IntList` из примера 9.2 не содержит поля `serialVersionUID`, его номер версии был неявно вычислен на основе API этого класса. Чтобы дать новой версии класса `IntList` номер, совпадающий с номером оригинальной версии, используется утилита *serialver*, входящая в состав Java SDK:

```
% serialver com.davidflanagan.examples.serialization.IntList
IntList:  static final long serialVersionUID = 4538804519406678841L;
```

Теперь можно запустить *serialver* и задать номер оригинальной версии класса. *serialver* печатает определение поля, пригодное для включения в модифицированную версию класса. Включив это константное поле в модифицированный класс, вы восстановите совместимость по сериализации между новой и первоначальной версией.

Дополнительные возможности учета версий

Иногда в класс вносятся исправления, изменяющие способ сохранения классом своего состояния. Вообразим класс `Rectangle`, представляющий прямоугольник как координаты верхнего левого угла плюс его ширина и высота. Предположим теперь, что класс был заново реализован так, что он сохраняет прежний открытый (public) интерфейс, но теперь прямоугольник представляется двумя точками: координатами верхнего левого и правого нижнего углов. Внутренние закрытые поля этого класса изменились, так что должно казаться, что совместимость

по сериализации между двумя реализациями этого класса попросту невозможна.

В Java 1.2 и более поздних версий, однако, механизм сериализации был усовершенствован, что позволило разделить формат сериализации и поля, используемые в конкретной реализации класса. Класс может теперь объявить закрытое поле `serialPersistentFields`, ссылающееся на массив объектов `java.io.ObjectStreamField`. Каждый из этих объектов определяет имя поля и тип поля. Эти поля не должны быть как-то связаны с полями, реализованными в классе; они являются полями сериализованной формы класса. Путем определения этого массива объектов `ObjectStreamField` класс задает свой формат сериализации. При определении новой версии класса эта новая версия должна быть способна сохранять и восстанавливать свое состояние в формате, определяемом массивом `serialPersistentFields`.

Приемы чтения и записи полей сериализации, объявленных массивом `serialPersistentFields`, не охвачены этой главой. Для получения дополнительной информации просмотрите методы `putFields()` и `writeFields()` класса `ObjectOutputStream` и метод `readFields()` класса `ObjectInputStream`. Посмотрите также примеры по дополнительным возможностям сериализации, приведенные в документации по Java SDK.

Сериализация апплетов

Особенно интересным применением сериализации объектов является сериализация апплетов (см. главу 15 «Апплеты»). При появлении Java 1.1 у HTML-тега <APPLET> появляется новый атрибут `OBJECT`, который можно использовать вместо атрибута `CODE` для задания файла сериализованного объекта вместо файла класса. Встретив такой тег <APPLET>, *визуализатор апплетов* или броузер создают апплет, десериализуя его.

Интересно это в силу того, что таким образом апплет может распространяться в заранее инициализированном состоянии. Код апплета даже может не содержать инициализирующий код. Для примера представим себе графическое средство разработки GUI, позволяющее программисту создавать GUI по технологии point-and-click («укажи-и-щелкни»). Такое средство разработки позволяет создать дерево AWT-компонентов в панели `Applet` и сериализовать апплет вместе со всеми компонентами GUI, которые он содержит. После десериализации апплет будет содержать полный GUI, несмотря на то что файл класса апплета не содержит никакого кода, создающего этот GUI.

Вы можете поэкспериментировать с сериализацией апплетов при помощи программы *appletviewer*. Запустите сначала апплет на *appletviewer* обычным способом. *appletviewer* загрузит апплет и запустит его методы `init()` и `start()`. Затем выберите в меню элемент **Stop**, чтобы ос-

тановить апплет. Теперь примените элемент меню **Save** для сериализации апплета в файл. По принятому соглашению вашему сериализованному апплету следует дать расширение *.ser*. Если апплет ссылается на какой-нибудь несериализуемый объект, сериализация может не удаться. Проблемы могут встретиться при сериализации апплета, использующего потоки исполнения (thread) или изображения (image).

Выполнив сериализацию апплета, создайте HTML-файл с подобным тегом <APPLET>:

```
<APPLET OBJECT="MyApplet.ser" WIDTH=400 HEIGHT=200></APPLET>
```

Наконец, примените *appletviewer* к этому новому HTML-файлу. Он должен десериализовать и отобразить апплет. Для апплета, созданного таким способом, метод апплета `init()` не вызывается (поскольку он вызывался до сериализации), но его метод `start()` вызывается (поскольку апплет должен был быть остановлен перед сериализацией).

Упражнения

9-1. Класс `java.util.Properties`, по существу, является хеш-таблицей, преобразующей строковые ключи в строковые значения. Он определяет метод `store()`, сохраняющий содержимое класса в байтовый поток, и метод `load()`, загружающий его оттуда. Эти методы, хотя и используют байтовый поток, сохраняют объект `Properties` в удобном для человеческого восприятия текстовом формате. Класс `Properties` наследует от интерфейса `Serializable` через свой родительский класс `java.util.Hashtable`. Определите подкласс `Properties`, реализующий методы `storeBinary()` и `loadBinary()`, использующие сериализацию объектов для сохранения и загрузки объекта `Properties` в двоичной форме. Вам может также понадобиться использовать в своих методах потоки `java.util.zip.GZIPOutputStream` и `java.util.zip.GZIPInputStream`, чтобы сделать ваш двоичный формат еще более компактным. Примените программу *serialver*, чтобы получить значение `serialVersionUID` для вашего класса.

9-2. Как отмечалось в предыдущем примере, объект `Properties` имеет методы `store()` и `load()`, позволяющие ему сохранять и восстанавливать свое состояние. Определите производный от `Externalizable` подкласс `Properties`, использующий методы `store()` и `load()` как основу для его методов `writeExternal()` и `readExternal()`. Чтобы все это заработало, метод `writeExternal()` должен определять количество байт, записанных методом `store()`, и записывать это значение первым, перед записываемыми байтами. Это позволит методу `readExternal()` узнать, когда перестать считывать. Включите также в используемый вами экстернализируемый формат данных номер версии.