2-е издание

Охватывает Java 2 версии 1.3

JAVA™ В ПРИМЕРАХ

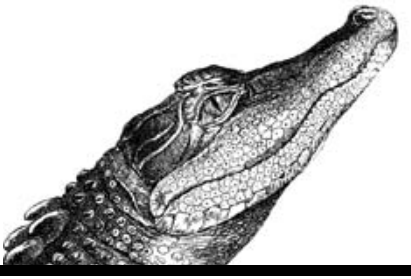
СПРАВОЧНИК

Учебное пособие к книге «Java. Справочник»

СИМВОЛ®

O'REILLY®

Дэвид Флэнаган



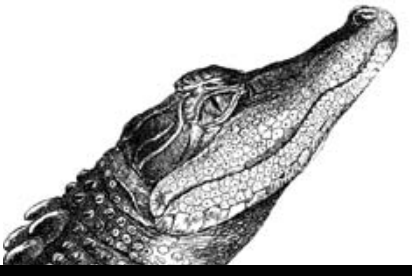
JAVA EXAMPLES IN A NUTSHELL

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

СИМВОЛ®

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

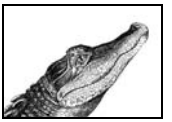
Второе издание книги «Java в примерах. Справочник» содержит 164 законченных практических примера: свыше 17 900 строк тщательно прокомментированного, профессионально написанного Java-кода, работающего с 20 различными программными интерфейсами Java, такими как сервлеты, JSP, XML, Swing и Java 2D. Приведены 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

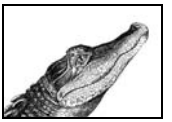
Июль 2000

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

I

Основные Java API

Часть I содержит примеры, демонстрирующие основные возможности Java и базовые программные интерфейсы Java. Эти примеры соответствуют той части языка, которая рассматривается в книге «Java in a Nutshell».



Глава 1

Основы Java

В этой главе содержатся примеры, демонстрирующие основы синтаксиса языка Java; подразумевается, что она изучается вместе с главой 2 книги «Java in a Nutshell»[1]. Если у вас есть существенный опыт программирования на C или C++, вы легко поймете материал этой главы. Если же вы переходите на Java с другого языка, вам, возможно, понадобится изучить эти примеры более тщательно.

Важнейшим шагом при изучении нового языка программирования является овладение основными управляющими операторами языка. В отношении Java это означает изучение оператора ветвления `if/else` и операторов цикла `while` и `for`. Учиться программировать хорошо – это как учиться решать словесные задачи на уроках алгебры в средней школе: вы должны перевести абстрактное описание задачи на конкретный математический язык (в нашем случае на язык Java). Когда вы научитесь мыслить категориями операторов `if`, `while` и `for`, другие операторы Java, такие как `break`, `continue`, `switch` и `try/catch/finally`, будет уже легко усвоить. Обратите внимание на то, что хотя Java – это объектно-ориентированный язык, мы не будем обсуждать объекты до главы 2.

Итак, приняв изложенное за введение и положив себе целью изучить основы синтаксиса, перейдем прямо к написанию программ на Java.

Hello World

Еще в 1978 году Брайан Керниган (Brian Kernighan) и Деннис Ричи (Dennis Ritchie) в своей классической книге «Язык программирования C»

---

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

писали: «Программа, которую следует написать первой, для всех языков одна». Разумеется, они имели в виду программу «Hello World». Реализация программы «Hello World» на языке Java показана в примере 1.1.

Пример 1.1. Hello.java

```
package com.davidflanagan.examples.basics;    // Уникальный префикс имени
                                              // класса
public class Hello {                          // В Java все является классом
    public static void main(String[] args) {  // Все программы должны
                                              // содержать метод main()
    System.out.println("Hello World!");       // Говорим Hello!
    }                                         // Отмечает конец main()
}                                             // Отмечает конец класса
```

Первой строкой программы является объявление package. Оно указывает имя *пакета* (*package*), в который входит эта программа. Именем программы (как мы это видим из второй строки) является Hello. Имя пакета – это com.davidflanagan.examples.basics. Мы можем соединить их, чтобы получить полное имя программы, com.davidflanagan.examples.basics.Hello. Применение пакетов обеспечивает каждую программу Java уникальным пространством имен. Поместив программу Hello в пакет, я гарантирую, что если еще кто-нибудь назовет свою программу Hello, конфликта имен не произойдет. Чтобы обеспечить заведомую уникальность имени пакета, я следую соглашению, по которому имя пакета начинается с доменного имени в Интернете, записанного в обратном порядке.[1] В каждой главе этой книги используется свой пакет с именем, начинающимся с префикса com.davidflanagan.examples. В этом примере, поскольку глава посвящена основам языка Java, именем пакета будет com.davidflanagan.examples.basics.

Ценность программы «Hello World» заключается в том, что она является шаблоном, который вы можете расширять в собственных экспериментах с Java. Вторая и третья строки в примере 1.1 являются обязательной составляющей шаблона. Любая программа – на самом деле, любой кусок кода на Java – является классом. Во второй строке программы говорится, что мы создаем класс Hello. Она также сообщает, что наш класс является внешним (public); это означает, что его может использовать кто угодно.

Всякая самостоятельная программа на Java обязательно содержит метод main(). Именно с этого места интерпетатор Java начинает исполнение программы, написанной на Java. В третьей строке примера объявляется метод main(). В ней говорится, что этот метод является открытым (public), что он не возвращает значения (или, другими словами,

[1] Мой веб-сайт называется *http://www.davidflanagan.com*, поэтому имя моего пакета начинается с com.davidflanagan.

возвращает значение void) и что в качестве аргументов он получает строковый массив. Имя этого массива – args. В этой строке утверждается также, что метод main() является статическим (static). (В этой главе мы будем иметь дело исключительно со статическими методами. В главе 2, когда мы начнем работать с объектами, вы узнаете, что такое статические методы и что методы, как правило, являются, напротив, нестатическими.)

Вы, во всяком случае, можете попросту заучить следующую строку:

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

Всякая самостоятельная программа на Java содержит строку, которая выглядит в точности, как эта. (По правде говоря, строковый массив вы можете назвать, как вам заблагорассудится, но обычно его называют именно args.)

Пятая и шестая строки примера 1.1 просто отмечают конец метода main() и класса Hello. Подобно большинству современных языков, Java является языком с блочной структурой. Это означает, что у таких элементов, как классы и методы, есть тело, являющееся «блоком» программного кода. В Java начало блока отмечается открывающей фигурной скобкой {, а конец – закрывающей }. Блоки методов всегда определяются внутри блоков классов и, как мы увидим из нижеследующих примеров, блоки методов могут содержать операторы if и циклы for, образующие внутри метода подблоки. Более того, такие блоки операторов могут вкладываться друг в друга на любую глубину.

Три первые и две последние строки примера 1.1 принадлежат каркасу приложения на Java. Интерес представляет только четвертая строка примера. Эта строка печатает слова «Hello World!». Метод System.out.println() отправляет выводимую строку в «стандартный вывод», которым обычно является экран. Этот метод применяется на протяжении этой и многих последующих глав этой книги. Но только в главе 3 «Ввод/вывод» вы поймете, что он делает в действительности. Если вам не терпится это узнать, найдите классы java.lang.System и java.io.PrintStream в книге «Java in a Nutshell» (или в каком-нибудь другом справочном руководстве по Java).

Последнее, о чем следует упомянуть в связи с этой программой, – это комментарии. В примере 1.1 применяются комментарии в стиле C++, которые начинаются с символов // и продолжаются до конца строки. Таким образом, все, что находится между символами // и концом строки, игнорируется компилятором Java. Вы обнаружите, что примеры в этой книге тщательно откомментированы. Комментарии в кодах заслуживают внимания, поскольку затрагивают некоторые вопросы, не нашедшие отражения в тексте.

Запуск программы «Hello World»

Чтобы запустить программу, нужно первым делом ее набрать.[1] Наберите при помощи текстового редактора программу `Hello` в том виде, в котором она приведена в примере 1.1. Но опустите теперь объявление пакета в первой строке. Сохраните программу в файле *Hello.java*.

Второй шаг состоит в компиляции программы. Если вы используете Java Software Development Kit (SDK) производства компании Sun, то будете компилировать программу командой *javac*.[2] Перейдите в каталог, в котором содержится файл *Hello.java*, и наберите эту команду (в предположении, что путь к *javac* известен системе):

```
% javac Hello.java
```

Если пакет Java SDK был правильно установлен, *javac* через небольшой промежуток времени создаст файл с именем *Hello.class*. В этом файле будет содержаться откомпилированная версия программы. Я уже говорил, что все, что вы пишете в Java, является классом, и этот факт отражен в расширении *.class* имени вашего файла. Важное правило, связанное с компиляцией программ на Java, состоит в том, что имя файла за вычетом расширения *.java* должно совпадать с именем определенного в файле класса. Таким образом, если бы вы набрали пример 1.1 и сохранили его в файле с названием *HelloWorld.java*, вам не удалось бы его откомпилировать.

Чтобы запустить программу (с применением Java SDK), наберите:

```
% java Hello
```

Эта команда произведет следующий вывод:

```
Hello World!
```

Команда *java* – это интерпретатор Java; он запускает виртуальную машину Java. Вы передаете *java* имя запускаемого класса. Обратите внимание на то, что при этом следует указывать имя класса `Hello`, а не имя файла *Hello.class*, содержащего скомпилированный класс.

Выше было показано, как компилируется и запускается программа на Java, не содержащая объявления пакета. Если вы опустили объявление `package` при наборе *Hello.java*, этих инструкций достаточно (если это не так, проверьте, правильно ли вы набрали программу). На прак-

---

[1] Хотя этот пример включен в онлайновый архив примеров, я советую вам набрать его самостоятельно. Если вы это сделаете, выражения языка Java глубже запечатлятся в вашей памяти. Я также буду предлагать вам модифицировать этот пример при объяснении некоторых аспектов исполнения программы.

[2] Если вы применяете другую среду разработки программ на Java, изучите инструкции производителя по компиляции и запуску программ и следуйте им.

тике, однако, все нетривиальные программы на Java (включая примеры из этой книги) содержат объявления package. При наличии объявления пакета процесс компиляции и запуска программ на Java несколько усложняется. Как я уже отмечал, программа на Java должна записываться в файл с именем, совпадающим с именем класса. Когда класс включается в пакет, добавляется требование, состоящее в том, чтобы имя каталога, в котором хранится этот файл, совпадало с именем пакета.

Давайте вернем в *Hello.java* объявление пакета:

```
package com.davidflanagan.examples.basics;
```

Создайте теперь каталог (или папку), в котором вы будете хранить все примеры из этой книги. В системе Windows, например, вы можете создать папку *c:\jenut2*. Под Linux можно создать каталог *~/jenut2*. В этом каталоге создайте подкаталог *com*. Теперь в каталоге *com* создайте подкаталог с именем *davidflanagan*, а затем создайте подкаталог *examples* в *davidflanagan*. Наконец, создайте в *examples* подкаталог *basics*. Скопируйте программу *Hello.java* (содержащую объявление пакета) в этот последний каталог. Под Windows получится файл с именем:

```
c:\jenut2\com\davidflanagan\examples\basics\Hello.java
```

Создав структуру каталогов и разместив в ней программу на Java, вы должны будете сообщить компилятору и интерпретатору Java, где теперь ее искать. Компилятору и интерпретатору нужно знать только выбранный вами базовый каталог; файл *Hello.class* они будут искать в его подкаталоге, ориентируясь по имени пакета. Чтобы указать Java направление поиска, вы должны присвоить значение переменной окружения CLASSPATH способом, соответствующим вашей операционной системе. Если, работая под Windows, вы приняли предложенное мной имя папки (*c:\jenut2*), можно использовать следующую команду:

```
C:\> set CLASSPATH=.;c:\jenut2
```

Эта команда предлагает Java искать программу сперва в текущем каталоге (.), а затем в каталоге *c:\jenut2*.

Под Unix при использовании оболочки *csh* можно написать следующую команду:

```
% setenv CLASSPATH .:/home/david/jenut2
```

В оболочках *sh* или *bash* соответствующая команда будет такой:

```
$ CLASSPATH=.:/home/david/jenut2; export CLASSPATH
```

Можно автоматизировать этот процесс, включив команду установки переменной CLASSPATH в стартовый файл, в *autoexec.bat* под Windows или в *.cshrc* под Unix (с оболочкой *csh*). Установив значение CLASSPATH, вы можете двинуться дальше и откомпилировать и запустить програм-

му `Hello`. Перед компиляцией перейдите в каталог *examples/basics*, содержащий *Hello.java*. Запустите компилятор, как раньше:

```
% javac Hello.java
```

Результатом будет файл *Hello.class*.

Для запуска программы следует, как прежде, вызвать интерпретатор Java, но на этот раз вам придется указать полностью квалифицированное имя программы, чтобы интерпретатору стало в точности известно, какую именно программу вы хотите исполнить:

```
% java com.davidflanagan.examples.basics.Hello
```

Установив переменную `CLASSPATH`, вы можете запускать интерпретатор Java из любого каталога вашей системы, и он всегда найдет нужную программу. Если набор вручную таких длинных имен утомляет, вы, возможно, изготовите для себя подходящий командный файл или сценарий, который будет выполнять это автоматически.

Заметьте, что таким же образом запускаются и исполняются все программы на Java, так что мы не будем повторять описание этих шагов. Один шаг, который вы сможете, разумеется, пропустить, – это набор программы. Вы можете скачать исходные коды примеров с сайта *http://www.davidflanagan.com/javaexamples2*.

FizzBuzz

Я научился играть в FizzBuzz очень давно, на уроках французского в начальной школе, где это помогало учиться счету по-французски. Игроки по очереди считают вслух от 1 и далее. Правила просты: на своем ходу вы называете очередное число. Но если это число кратно пяти, вместо него следует произнести слово «fizz» (желательно с французским акцентом). Если число кратно семи, надо сказать «buzz». А если число делится и на семь, и на пять, нужно произнести «fizzbuzz». Ошибившийся вылетает, и игра продолжается без него.

Пример 1.2 – это программа на Java под названием `FizzBuzz`, которая играет в эту игру. Конечно, это не самый интересный вариант игры, так как компьютер играет сам с собой и к тому же считает не по-французски! Нам будет интересен Java-код, содержащийся в этом примере. В нем демонстрируется применение цикла `for` для счета от 1 до 100 и используются операторы `if/else` для принятия решения о том, что следует вывести: число либо «fizz», либо «buzz», либо «fizzbuzz». (В этом примере оператор `if/else` появляется в форме оператора `if/elseif/elseif/else`, что мы вкратце обсудим позже.)

В этой программе используется метод `System.out.print()`. Он отличается от метода `System.out.println()` только тем, что не заканчивает выводимую строку. Все, что выводится далее, выводится в той же строке.

В этом примере показан еще один стиль представления комментариев. Любой набор строк между парой символов /* и парой символов */ является в Java комментарием и игнорируется компилятором. Когда комментарии начинаются с последовательности /**, как это имеет место в одном из комментариев примера, этот текст является еще и *doc-комментарием*. Это означает, что его содержимое используется программой *javadoc*, автоматически создающей API-документацию на основании исходных кодов Java.

Пример 1.2. FizzBuzz.java

```
package com.davidflanagan.examples.basics;
/**
 * Эта программа играет в FizzBuzz. Она считает до 100, заменяя каждое число,
 * кратное 5, словом «fizz», каждое число, кратное 7, - словом «buzz»
 * и каждое число, кратное 35, - словом «fizzbuzz». Для определения того,
 * делится ли одно число на другое, в ней используется
 * оператор остатка целочисленного деления (%).
 **/
public class FizzBuzz {                      // В Java все является классом
  public static void main(String[] args) {  // Каждая программа содержит main()
    for(int i = 1; i <= 100; i++) {          // Считаем от 1 до 100
      if (((i % 5) == 0) && ((i % 7) == 0)) // Делится ли число и на 5, и на 7?
        System.out.print("fizzbuzz");
      else if ((i % 5) == 0)                // Делится ли число на 5?
        System.out.print("fizz");
      else if ((i % 7) == 0)                // Делится ли число на 7?
        System.out.print("buzz");
      else System.out.print(i);            // Число не делится ни на 5, ни на 7
      System.out.print(" ");
    }
    System.out.println();
  }
}
```

Операторы `for` и `if/else` следует пояснить для тех программистов, которые с ними не встречались. Оператор `for` описывает цикл, в котором некоторый фрагмент кода выполняется несколько раз. За ключевым словом `for` следуют три выражения языка Java, определяющие параметры цикла. Синтаксис определения цикла таков:

```
for(инициализация ; проверка ; обновление)
тело
```

В выражении *инициализация* осуществляется необходимая инициализация. Оно исполняется только однажды, перед началом цикла. Здесь обычно устанавливается начальное значение переменной цикла. Часто, как и в этом примере, счетчик цикла используется только в цикле, так что в выражении *инициализация* переменная также и объявляется.

В выражении *проверка* проверяется, следует ли продолжать цикл. Это выражение вычисляется перед каждым исполнением тела цикла. Если его значением оказывается `true` (истинно), цикл исполняется. Если

же его значением оказывается `false` (ложно), тело цикла не исполняется, и цикл прекращается.

Выражение *обновление* вычисляется после каждой итерации цикла; оно осуществляет все необходимое для следующей итерации цикла. Обычно оно просто увеличивает или уменьшает значение счетчика цикла.

Наконец, *тело* – это код Java, исполняющийся в цикле. Он может представлять собой одиночный оператор Java или целый блок кода Java, заключенный в фигурные скобки.

Из этого объяснения становится ясно, что цикл `for` в примере 1.2 считает от 1 до 100.

Оператор `if/else` проще оператора `for`. Его синтаксис таков:

```
if (выражение)
    оператор1
else
    оператор2
```

Когда интерпретатор Java встречает оператор `if`, он вычисляет значение *выражения*. Если оно оказывается `true`, исполняется *оператор1*. В противном случае исполняется *оператор2*. Вот и все, что делает `if/else`; здесь нет никакого зацикливания, так что выполнение программы продолжается с первого оператора, следующего за `if/else`. Оператор `else` и следующий за ним *оператор2* совершенно необязательны. Если они отсутствуют и значение *выражения* оказывается равным `false`, оператор `if` не делает ничего. Операторы, следующие за ключевыми словами `if` и `else`, могут быть как одиночными операторами Java, так и целыми блоками кода Java, заключенными в фигурные скобки.

Что следует отметить в связи с операторами `if/else` (и операторами `for`, кстати говоря), так это то, что они могут содержать другие операторы, в частности другие операторы `if/else`. Именно так этот оператор используется в примере 1.2, где мы видим нечто похожее на оператор `if/elseif/elseif/else`. На самом деле это просто оператор `if/else`, находящийся внутри оператора `if/else`, заключенного в другом операторе `if/else`. Эта структура проясняется, если переписать код с применением фигурных скобок:

```
if (((i % 5) == 0)&& ((i % 7) == 0))
  System.out.print("fizzbuzz");
else {
  if ((i % 5) == 0)
    System.out.print("fizz");
  else {
    if ((i % 7) == 0)
      System.out.print("buzz");
    else
      System.out.print(i);
  }
}
```

Заметим, однако, что подобные вложенные операторы `if/else` обычно записываются без таких фигурных скобок. Программная конструкция `else if` – это широко используемый прием, к которому вы скоро привыкнете. Вы, возможно, заметили также, что я придерживаюсь компактного стиля при программировании, стараясь по возможности все помещать в одной строке. Например, вам часто придется видеть следующее:

```
if (выражение) оператор
```

В таком виде программа компактнее и удобнее в обращении, поэтому ее легче изучать в напечатанном виде. В своих кодах вы, возможно, предпочтете более структурированный, менее компактный стиль.

Числа Фибоначчи

Числа Фибоначчи – это последовательность чисел, в которой каждое следующее число равно сумме двух предыдущих. Начало этой последовательности – 1, 1, 2, 3, 5, 8, 13, и ее можно бесконечно продолжать. Эта последовательность встречается в природе в самых неожиданных местах. Например, число лепестков у большинства видов цветов является одним из чисел Фибоначчи.

Пример 1.3 представляет собой программу, которая вычисляет и отображает 20 первых чисел ряда Фибоначчи. По поводу этой программы следует сказать несколько слов. Во-первых, в ней снова используется оператор `for`. В нем также объявляются и используются переменные для хранения двух последних чисел последовательности, так что их можно сложить, чтобы получить следующее число.

Пример 1.3. Fibonacci.java

```
package com.davidflanagan.examples.basics;
/**
 * Эта программа распечатывает 20 первых чисел ряда Фибоначчи.
 * Каждый его элемент формируется как сумма двух предыдущих элементов ряда,
 * начиная с элементов 1 и 1.
 **/
public class Fibonacci {
 public static void main(String[] args) {
  int n0 = 1, n1 = 1, n2;        // Инициализация переменных
  System.out.print(n0 + " " +    // Печать первого и второго элементов
       n1 + " ");                // ряда
  for(int i = 0; i < 18; i++) { // Цикл для следующих 18 элементов
   n2 = n1 + n0;           // Следующий элемент является суммой двух предыдущих
   System.out.print(n2 + " ");  // Распечатать
   n0 = n1;       // Первый с конца становится вторым с конца
   n1 = n2;       // а текущее число становится последним
  }
  System.out.println();          // Завершаем строку
 }
}
```

Использование аргументов командной строки

Как мы уже видели, всякая самостоятельная программа на Java должна содержать объявление метода, в точности соответствующее следующей сигнатуре:

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

Эта сигнатура говорит о том, что методу `main()` передается массив строк. Что это за строки и откуда они берутся? Массив `args` содержит все аргументы, передаваемые интерпретатору Java в командной строке вслед за именем запускаемого на выполнение класса. Пример 1.4 представляет собой программу `Echo`, читающую и распечатывающую эти аргументы. Вы могли бы вызвать эту программу так:

```
% java com.davidflanagan.examples.basics.Echo this is a test
```

Программа отвечает:

```
this is a test
```

В этом случае длина массива `args` равна четырем. Первый элемент массива `args[0]` — это строка «this», а последний элемент массива `args[3]` — это строка «test». Как видите, массивы в Java начинаются с элемента `0`. Если вы до этого программировали на языке, в котором нумерация элементов массивов начинается с 1, вам придется как-то к этому привыкнуть. В частности, вы должны запомнить, что если длина массива `a` равна `n`, последним элементом массива будет `a[n-1]`. Вы можете определить длину массива, приписав к его имени `.length`, как это показано в примере 1.4. Этот пример демонстрирует также применение цикла `while`. Цикл `while` — это упрощенная форма цикла `for`; в нем вам придется самостоятельно производить инициализацию и обновление счетчика. Большинство циклов `for` могут быть переписаны как циклы `while`, но компактный синтаксис цикла `for` делает его более распространенным. В этом примере цикл `for` вполне подошел бы и был бы даже предпочтительнее.

Пример 1.4. Echo.java

```
package com.davidflanagan.examples.basics;
/**
 * Эта программа распечатывает все заданные в ее командной строке аргументы.
 **/
public class Echo {
 public static void main(String[] args) {
  int i = 0;                        // Инициализация переменной цикла
  while(i < args.length) {          // Цикл до конца массива
   System.out.print(args[i] + " "); // Печать каждого из аргументов
   i++;                             // Увеличение переменной цикла
  }
```

```
    System.out.println();              // Переход на следующую строку
  }
}
```

Эхо-вывод в обратном порядке

Пример 1.5 очень похож на программу Echo из примера 1.4 за исключением того, что в нем аргументы командной строки выводятся в обратном порядке, как и символы в каждом из аргументов. Таким образом, программа Reverse, вызванная способом, представленным ниже, произведет следующий вывод:

```
% java com.davidflanagan.examples.basics.Reverse this is a test
tset a si siht
```

Эта программа интересна тем, что ее вложенные циклы for считают не вперед, а назад. Интересно в ней также то, что она манипулирует строковыми объектами (String), вызывая методы этих объектов, и синтаксис становится несколько сложнее. Рассмотрим в качестве примера центральное выражение из этой программы:

```
args[i].charAt(j).
```

Это выражение первым делом выделяет i-й элемент массива args[]. Из объявления массива в сигнатуре метода main() мы знаем, что это массив строк (String array) и что он, следовательно, состоит из строковых объектов (String objects). (В языке Java String это не примитивный тип, такой как целые и логические значения: строки являются полноценными объектами.) Выделив из массива i-ю строку, мы вызываем принадлежащий этому объекту метод charAt() с аргументом j. (Символ «.» в этом выражении указывает на то, что вы имеете дело с методом или с полем объекта.) Как можно догадаться из названия метода (и при желании убедиться, заглянув в справочное руководство), он выделяет указанный символ из объекта String. Таким образом, это выражение извлекает j-й символ из i-го аргумента командной строки. Вооружившись этими знаниями, вы сумеете до конца разобраться в примере 1.5.

Пример 1.5. Reverse.java

```
package com.davidflanagan.examples.basics;
/**
 * Эта программа выводит «от конца к началу» аргументы, заданные
 * в командной строке.
 **/
public class Reverse {
 publicstatic void main(String[] args) {
  // Цикл проходит массив аргументов от конца к началу
  for(int i = args.length-1; i >= 0; i--) {
   // Цикл проходит от конца к началу символы в каждом аргументе
   for(int j=args[i].length()-1; j>=0; j--) {
    // Печатается символ j аргумента i.
```

```
      System.out.print(args[i].charAt(j));
     }
     System.out.print(" "); // После каждого аргумента выводится пробел
    }
    System.out.println();  // И, закончив, переходим на следующую строку
   }
  }
```

FizzBuzz с оператором switch

Пример 1.6 представляет другую версию игры FizzBuzz. В ней для определения того, что следует выводить для каждого числа, вместо вложенных операторов if/else используется оператор switch. Сначала посмотрите на пример, а затем прочитайте объяснение того, как работает оператор switch.

Пример 1.6. FizzBuzz2.java

```
package com.davidflanagan.examples.basics;
/**
 * Этот класс очень похож на класс FizzBuzz, но в нем вместо
 * повторяющихся операторов if/else применяется оператор switch
 **/
public class FizzBuzz2 {
 public static void main(String[] args) {
  for(int i = 1; i <= 100; i++) {  // Считаем от 1 до 100
   switch(i % 35) {                // Каков остаток при делении на 35?
   case 0:                         // Для кратных 35...
    System.out.print("fizzbuzz "); // Печать «fizzbuzz»
    break;                         // Не пропустите этот оператор!
   case 5: case 10: case 15:       // Если остаток – один из представленных,
   case 20: case 25: case 30:      // число кратно 5,
    System.out.print("fizz ");     // и печатается «fizz»
    break;
   case 7: case 14: case 21: case 28: // Для всех кратных 7...
    System.out.print("buzz ");        // печатается «buzz»
    break;
   default:                      // Для любого другого числа...
    System.out.print(i + " ");   // печатается число
    break;
   }
  }
  System.out.println();
 }
}
```

Оператор switch работает как диспетчер на маневровой площадке, из множества вариантов выбирающий для поезда (программы) подходящий путь (фрагмент кода). Оператор switch часто может служить альтернативой повторяющимся операторам if/else, но он работает, только когда тестируемое значение принадлежит примитивному целочис-

ленному типу (другими словами, если оно не double и не String) и когда его значение сравнивается с константами. Синтаксис оператора switch таков:

```
switch(выражение) {
 операторы
}
```

За оператором switch следуют *выражение* в скобках и блок кода в фигурных скобках. После вычисления *выражения* оператор switch исполняет определенную часть кода в скобках, зависящую от значения выражения. Как оператор switch узнает, где начинается код, соответствующий определенному значению? Эта информация обозначается метками case: и специальной меткой default:. За каждой меткой case: следует некоторое целочисленное значение. Если результат вычисления *выражения* оказывается равным этому значению, оператор switch начинает исполнять код, непосредственно следующий за меткой case:. Если не найдена метка case:, соответствующая значению *выражения*, оператор switch исполняет код, следующий за меткой default:, если таковая имеется. Если метки default: нет, switch не делает ничего.

Оператор switch необычен тем, что в нем меткам case: не соответствуют блоки кода. Вместо этого метки case: и default: просто отмечают точки входа в один большой блок кода. Обычно за каждой меткой следуют несколько операторов и затем оператор break, вызывающий передачу управления за пределы оператора switch. Если вы пропустите оператор break в конце кода, соответствующего метке, исполнение «провалится» в код, относящийся к следующей метке. Если вы хотите посмотреть на это в действии, уберите оператор break из примера 1.6 и посмотрите, что получится, когда вы запустите программу. Пропуск оператора break в операторе switch – распространенный источник ошибок.

Вычисление факториалов

Факториал целого числа получается как произведение этого числа на все положительные целые, которые меньше него. Таким образом, факториал 5 (записывается 5!) – это произведение 5*4*3*2*1, или 120. Пример 1.7 определяет класс Factorial, содержащий метод factorial(), вычисляющий факториалы. Этот класс не является самостоятельной программой, но определенный в нем метод может использоваться любой другой программой. Сам по себе метод очень прост; в следующих разделах мы увидим несколько его вариантов. В качестве упражнения вы могли бы придумать, как переписать этот пример с использованием цикла while вместо цикла for.

Пример 1.7. Factorial.java

```
package com.davidflanagan.examples.basics;
/**
```

```
 * Этот класс не определяет метод main() и поэтому не является
 * самостоятельной программой. Он, тем не менее, определяет полезный
 * метод, который можно использовать в других программах.
 **/
public class Factorial {
 /** Вычисляем и возвращаем x!, факториал x */
 public static int factorial(int x) {
  if (x < 0) throw new IllegalArgumentException("x должен быть >= 0");
  int fact = 1;
  for(int i = 2; i <= x; i++) // Цикл
   fact *= i;                 // Краткая запись для fact = fact * i;
  return fact;
 }
}
```

Рекурсивные факториалы

Пример 1.8 демонстрирует еще один способ вычисления факториалов. В нем используется прием программирования, называемый *рекурсией*. Рекурсия имеет место, когда метод вызывает сам себя, или, другими словами, рекурсивно обращается к себе. Рекурсивный алгоритм вычисления факториалов основывается на том факте, что `n!` равняется `n*(n-1)!`. Такой способ вычисления факториалов является классическим примером рекурсии. В этом случае этот прием не особенно эффективен, но у рекурсии есть множество важных применений, и этот пример доказывает, что ее применение в Java совершенно законно. В этом примере также вместо типа данных `int`, являющегося 32-битовым целым, используется тип данных `long`, являющийся 64-битовым целым. Факториалы быстро становятся очень большими, так что дополнительная емкость типа `long` делает метод `factorial()` более полезным.

Пример 1.8. Factorial2.java

```
package com.davidflanagan.examples.basics;
/**
 * Этот класс демонстрирует рекурсивный способ вычисления факториалов. Этот
 * метод многократно вызывает сам себя, основываясь на формуле: n! = n * (n-1)!
 **/
public class Factorial2 {
 public static long factorial(long x) {
  if (x < 0) throw new IllegalArgumentException("x должен быть >= 0");
  if (x <= 1) return 1;           // Здесь рекурсия прекращается
  else return x * factorial(x-1); // Шаг рекурсии - вызов самого себя
 }
}
```

Кэширование факториалов

Пример 1.9 предъявляет усовершенствованный вариант предыдущих примеров, вычисляющих факториалы. Факториалы – это идеальные кандидаты для кэширования, поскольку их вычисление требует времени и, что еще важнее, вы можете вычислить не так уж много факториалов в силу ограниченности типа данных long. Поэтому в этих примерах, раз уж факториал вычислен, его значение сохраняется для будущего употребления.

Помимо демонстрации приемов кэширования в этом примере вводится кое-что новое. Во-первых, в классе Factorial3 определяются статические поля:

```
static long[] table = new long[21];
static int last = 0;
```

Статические поля – это род переменных, сохраняющих, однако, свои значения в промежутках между вызовами метода factorial(). Это означает, что в статических полях могут сохраняться значения, вычисленные при одном вызове метода, для использования при другом его вызове.

Во-вторых, в этом примере можно увидеть, как создаются массивы:

```
static long[] table = new long[21];
```

Первая половина этой строки (перед знаком =) объявляет статическое поле table, которое будет массивом значений типа long. Вторая половина этой строки создает массив из 21 значения типа long при помощи оператора new.

Наконец, в этом примере показано, как генерируются исключения:

```
throw new IllegalArgumentException("Переполнение: x слишком велик.");
```

Исключения – это вид объектов Java. Они, как и массивы, создаются при помощи ключевого слова new. Если программа генерирует объект-исключение при помощи оператора throw, это означает, что возникло неожиданное обстоятельство или ошибка. Когда возникает исключение, управление в программе передается ближайшей секции catch оператора try/catch. Эта секция должна содержать код для обработки исключений. Если исключение не будет перехвачено, исполнение программы прекращается с сообщением об ошибке.

В примере 1.9 возникает исключение, уведомляющее вызывающую процедуру о том, что переданный ею аргумент слишком велик или слишком мал. Аргумент слишком велик, если он больше 20, поскольку мы не можем вычислять факториалы, превышающие 20!. Аргумент слишком мал, если он меньше 0, так как факториалы определены только для неотрицательных целых чисел. Как перехватывать и обрабатывать исключения, показано в следующих примерах этой главы.

Пример 1.9. Factorial3.java

```
package com.davidflanagan.examples.basics;
/**
 * Этот класс вычисляет факториалы и кэширует результаты в таблице
 * для дальнейшего употребления. 20! - самый большой факториал,
 * который мы можем вычислить с применением типа данных long,
 * поэтому проверим аргумент и выдадим исключение, если аргумент
 * окажется слишком большим или слишком маленьким.
 **/
public class Factorial3 {
 // Создаем массив для хранения факториалов от 0! до 20!.
 static long[] table = new long[21];
 // «Статический инициализатор»: инициализируем первый элемент массива
 static { table[0] = 1; } // Факториал 0 равен 1.
 // Запоминаем номер последнего вычисленного факториала
 static int last = 0;
 public static long factorial(int x) throws IllegalArgumentException {
  // Проверяем, не слишком ли мал или велик x. Выдаем исключение,
  // если это оказывается так
  if (x >= table.length) // ".length" возвращает длину любого массива
   throw new IllegalArgumentException("Переполнение: x слишком велик.");
  if (x<0) throw new IllegalArgumentException("x должен быть неотрицательным.");
  // Вычисляем и кэшируем все пока еще несохраненные значения
  while(last < x) {
   table[last + 1] = table[last] * (last + 1);
   last++;
  }
  // Возвращаем кэшированный факториал x
  return table[x];
 }
}
```

Вычисление факториалов больших чисел

В предыдущем разделе мы узнали, что 20! — это самый большой факториал, помещающийся в 64-битовом целом. Но что делать, если нужно вычислить 50! или 100!? Класс java.math.BigInteger представляет сколь угодно большие целые числа и обеспечивает методы для арифметических операций над ними. Пример 1.10 использует класс BigInteger для вычисления факториалов любого размера. Он содержит также простой метод main(), представляющий собой самостоятельную программу для тестирования метода factorial(). Эта тестовая программа может, например, сообщить, что 50! — это следующее 65-значное число:

```
30414093201713378043612608166064768844377641568960512000000000000
```

Пример 1.10 знакомит вас с оператором import. Этот оператор появляется в начале файла Java до определения какого-либо класса (но после объявления package). Он позволяет сообщить компилятору, какие

классы используются в программе. После того как, например, класс `java.math.BigInteger` импортирован, вам уже не нужно писать его полное имя; вы можете ссылаться на него просто как на `BigInteger`. Вы можете импортировать также целый пакет классов, включив в файл, например, такую строку:

```
import java.util.*
```

Обратите внимание на то, что классы пакета `java.lang` автоматически импортированы, поскольку они принадлежат текущему пакету, которым в нашем случае является `com.davidflanagan.examples.basics`.

В примере 1.10 применяется та же техника кэширования, что и в примере 1.9. Однако поскольку множество факториалов, которые могут быть вычислены, теперь не ограничено сверху, для кэширования нельзя применить массив фиксированного размера. Вместо этого в примере используется `java.util.ArrayList` — служебный класс, реализующий структуру данных, подобную массивам и способную разрастаться до любого нужного размера. Поскольку `ArrayList` — это объект, а не массив, при работе с ним приходится применять методы, такие как `size()`, `add()` и `get()`. Точно так же `BigInteger` — это объект, а не примитивное значение, поэтому для умножения объектов `BigInteger` нельзя применять оператор `*`. Вместо этого используется метод `multiply()`.

Пример 1.10. Factorial4.java

```
package com.davidflanagan.examples.basics;
// Импортируем классы, которые намереваемся использовать в нашей программе.
// Импортировав класс, мы уже не обязаны называть его полным именем.
import java.math.BigInteger; // Импортируем класс BigInteger из пакета java.math
import java.util.*;          // Импортируем все классы (включая ArrayList)
                             // пакета java.util
/**
 * В этой версии программы применяются целые числа произвольно большого
 * размера, поэтому вычисляемые значения не ограничены сверху.
 * Для кэширования вычисленных значений в ней применяется объект ArrayList
 * вместо массивов фиксированного размера. ArrayList похож на массив,
 * но может разрастаться до любого размера. Метод factorial() объявлен
 * как «synchronized», поэтому его можно смело использовать в
 * многопоточных программах. При изучении этого класса познакомьтесь
 * с java.math.BigInteger и java.util.ArrayList. Работая с версиями Java,
 * предшествующими Java 1.2, используйте Vector вместо ArrayList.
 **/
public class Factorial4 {
 protected static ArrayList table = new ArrayList(); // Создаем кэш
 static {  // Первый элемент кэша инициализируется значением 0! = 1
 table.add(BigInteger.valueOf(1));
 }
 /** Метод factorial(), использующий объекты BigInteger,
     сохраняемые в ArrayList */
 public static synchronized BigInteger factorial(int x) {
```

```
 if (x<0) throw new IllegalArgumentException("x должен быть неотрицательным.");
 for(int size = table.size(); size <= x; size++) {
  BigInteger lastfact = (BigInteger)table.get(size-1);
  BigInteger nextfact = lastfact.multiply(BigInteger.valueOf(size));
  table.add(nextfact);
 }
 return (BigInteger) table.get(x);
 }
 /**
 * Простой метод main(), который можно использовать в качестве
 * самостоятельной тестирующей программы для нашего метода factorial().
 **/
 public static void main(String[] args) {
 for(int i = 0; i <= 50; i++)
  System.out.println(i + "! = " + factorial(i));
 }
}
```

Обработка исключений

Пример 1.11 демонстрирует программу, использующую метод `Integer.parseInt()` для преобразования в число строки символов, заданной в командной строке. Затем программа вычисляет и печатает факториал этого числа, используя метод `Factorial4.factorial()`, определенный в примере 1.10. Пока все очень просто; это потребовало всего две строки кода. Оставшаяся часть примера занимается обработкой исключений, другими словами, заботится обо всем, что может пойти не так, как надо. Для обработки исключений в Java используется оператор `try/catch`. В его секции `try` содержится блок кода, который может выдать исключение. За ним может следовать любое число секций `catch`. Код в каждой секции `catch` заботится об определенном типе исключений.

В примере 1.11 возможны три ошибки пользовательского ввода, способные помешать нормальному выполнению программы. Поэтому две главные строки кода программы «обернуты» в секцию `try`, за которой следуют три секции `catch`. Каждая из последних уведомляет пользователя об определенной ошибке, печатая соответствующее сообщение. Этот пример очень прозрачен. Вы можете обратиться к главе 2 в «Java in a Nutshell», где исключения объясняются более подробно.

Пример 1.11. FactComputer.java

```
package com.davidflanagan.examples.basics;
/**
 * Эта программа вычисляет и отображает факториал числа,
 * заданного в командной строке. При помощи оператора try/catch
 * в ней обрабатываются возможные ошибки пользовательского ввода.
 **/
public class FactComputer {
 public static void main(String[] args) {
```

```
    // Пробуем вычислить факториал. Если что-то оказывается не в порядке,
    // обрабатываем исключения при помощи секции catch.
    try {
     int x = Integer.parseInt(args[0]);
     System.out.println(x + "! = " + Factorial4.factorial(x));
    }
    // Пользователь забыл задать аргумент.
    // Исключение возникает, если args[0] остается неопределенным.
    catch (ArrayIndexOutOfBoundsException e) {
     System.out.println("Вы должны задать аргумент");
     System.out.println("Формат: java FactComputer <число>");
    }
    // Аргумент не является числом. Выдается методом Integer.parseInt().
    catch (NumberFormatException e) {
     System.out.println("Задаваемый аргумент должен быть целым числом");
    }
    // Аргумент < 0. Выдается методом Factorial4.factorial()
    catch (IllegalArgumentException e) {
     // Отображает сообщение, выданное методом factorial():
     System.out.println("Плохой аргумент: " + e.getMessage());
    }
    }
   }
```

Интерактивный ввод

Пример 1.12 показывает еще одну программу для вычисления факториалов. В отличие от примера 1.11, однако, эта программа не останавливается после вычисления одного факториала. Вместо этого она предлагает пользователю ввести число, считывает это число, печатает его факториал и затем возвращается в начало и предлагает пользователю ввести новое число. Интереснее всего в этой программе техника, применяемая для считывания пользовательского ввода с клавиатуры. Для этого в ней используется метод `readLine()` объекта `BufferedReader`. Строка, в которой создается `BufferedReader`, может показаться запутанной. Примите пока на веру, что это работает; до главы 3 вам необязательно понимать, как это работает. В примере 1.12 следует также обратить внимание на применение метода `equals()` к объекту `line` типа `String` для проверки, не набрал ли пользователь «quit».

Код для анализа пользовательского ввода, вычисления и печати факториала такой же, как в примере 1.11, и снова он заключен в секцию `try`. В примере 1.12, однако, есть только одна секция `catch` для обработки возможных исключений. Она обрабатывает все объекты исключений, принадлежащих типу `Exception`. `Exception` — это суперкласс исключений всех типов, поэтому вызывается одна секция `catch` вне зависимости от типа возникшего исключения.

Пример 1.12. FactQuoter.java

```
package com.davidflanagan.examples.basics;
import java.io.*; // Импортируем все классы пакета java.io.,
                  // избавляя, тем самым, себя от лишнего набора.
/**
 * Эта программа отображает факториалы чисел,
 * вводимых пользователем в интерактивном режиме
 **/
public class FactQuoter {
 public static void main(String[] args) throws IOException {
 // Так подготавливается чтение строк, вводимых пользователем
 BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
 // Бесконечный цикл
 for(;;) {
  // Отображается подсказка для пользователя
  System.out.print("FactQuoter> ");
  // Считывается введенная пользователем строка
  String line = in.readLine();
  // Если считан конец файла,
  // или если пользователь набрал «quit», то конец
  if ((line == null) || line.equals("quit")) break;
  // Пытаемся проанализировать строку, вычислить и напечатать факториал
  try {
  int x = Integer.parseInt(line);
  System.out.println(x + "! = " + Factorial4.factorial(x));
  }
  // Если что-то не в порядке, отображается общее сообщение об ошибке
  catch(Exception e) { System.out.println("Invalid Input"); }
 }
 }
}
```

Применение объекта StringBuffer

Относительно класса `String`, используемого в Java для представления строк, следует отметить тот факт, что его объекты неизменяемы. Другими словами, не существует методов, которые позволяют изменить содержимое строки. Работающие со строками методы возвращают новую строку, а не преобразованную копию старой. Если вы хотите отредактировать строку по месту, вам придется воспользоваться объектом `StringBuffer`.

Пример 1.13 демонстрирует применение объекта `StringBuffer`. В этом примере в интерактивном режиме считывается строка пользовательского ввода, как в примере 1.12, и для хранения строки создается объект `StringBuffer`. Затем, применяя подстановочный шифр *rot13*, который просто «сдвигает по кругу» каждую букву алфавита на 13 мест (за Z следует A), программа кодирует каждый символ строки. За счет использования объекта `StringBuffer` можно поочередно заменять все сим-

волы строки. Сеанс с программой `Rot13Input` может выглядеть примерно так:

```
% java com.davidflanagan.examples.basics.Rot13Input
> Hello there. Testing, testing!
Uryyb gurer. Grfgvat, grfgvat!
> quit
%
```

Метод `main()` из примера 1.13 вызывает другой метод, `rot13()`, для осуществления непосредственно шифрования символа. Этот метод демонстрирует применение примитивного типа `char` языка Java и символьных литералов (то есть символов, заключенных в апострофы и используемых в программе буквально).

Пример 1.13. Rot13Input.java

```
package com.davidflanagan.examples.basics;
import java.io.*; // Мы будем вводить данные, поэтому импортируем
                  // классы ввода/вывода
/**
 * Эта программа читает вводимые пользователем строки, кодирует их
 * при помощи тривиального подстановочного шифра «Rot13»
 * и затем печатает закодированную строку.
 **/
public class Rot13Input {
 public static void main(String[] args) throws IOException {
 // Подготовка к чтению вводимых пользователем строк
 BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
 for(;;) {                       // Бесконечный цикл
  System.out.print("> ");        // Печать подсказки
  String line = in.readLine();   // Чтение строки
  if ((line == null) || line.equals("quit"))  // Если конец файла или «quit»...
  break;                                      // ...выход из цикла
  StringBuffer buf = new StringBuffer(line); // Применение StringBuffer
  for(int i = 0; i < buf.length(); i++)      // Для каждого символа...
  buf.setCharAt(i, rot13(buf.charAt(i)));    // ...прочитать, закодировать,
                                             // сохранить
  System.out.println(buf);          // Печатается закодированная строка
 }
 }

/**
 * Этот метод осуществляет подстановочное шифрование Rot13.
 * Он «сдвигает по кругу» каждую букву алфавита на 13 мест. Поскольку
 * в латинском алфавите 26 букв, этот метод и кодирует, и декодирует.
 **/
 public static char rot13(char c) {
 if ((c >= 'A') && (c <= 'Z')) { // Для заглавных букв
  c += 13;                       // Сдвигает вперед на 13
  if (c > 'Z') c -= 26;          // и вычитает при необходимости 26
 }
 if ((c >= 'a') && (c <= 'z')) { // То же самое для строчных букв
```

```
    c += 13;
    if (c > 'z') c -= 26;
   }
   return c;                        // Возвращает преобразованную букву
   }
  }
```

Сортировка чисел

Пример 1.14 реализует простой (но неэффективный) алгоритм сортировки массива чисел. В этом примере не вводятся новые элементы синтаксиса Java, но он интересен, поскольку в нем достигается уровень сложности, свойственный реальным программам. Алгоритм сортировки манипулирует элементами массива с применением оператора `if` в цикле `for`, заключенном в другой цикл `for`. Не пожалейте времени на внимательное изучение этого примера. Убедитесь, что вы точно поняли, как он сортирует массив чисел.

Пример 1.14. SortNumbers.java

```
package com.davidflanagan.examples.basics;
/**
 * Этот класс демонстрирует сортировку чисел при помощи простого алгоритма
 **/
public class SortNumbers {
   /**
   * Это очень простой алгоритм сортировки, не очень эффективный
   * при сортировке больших наборов элементов
   **/
   public static void sort(double[] nums) {
     // Цикл по всем элементам массива, в ходе которого
     // осуществляется сортировка.
     // На каждом шаге среди оставшихся неотсортированными
     // отыскиваем наименьший элемент и перемещаем
     // на первую неотсортированную позицию в массиве.
     for(int i = 0; i < nums.length; i++) {
         int min = i; // Хранит индекс наименьшего элемента
         // Находим наименьший элемент от i до конца массива
         for(int j = i; j < nums.length; j++) {
             if (nums[j] < nums[min]) min = j;
         }
         // Меняем местами наименьший элемент с элементом i.
         // Элементы между 0 и i остаются при этом отсортированными.
         double tmp;
         tmp = nums[i];
         nums[i] = nums[min];
         nums[min] = tmp;
     }
 }
```

```
    /** Это простая тестирующая программа для вышеприведенного алгоритма */
    public static void main(String[] args) {
        double[] nums = new double[10];    // Создается массив для хранения чисел
        for(int i = 0; i < nums.length; i++) // Генерируются случайные числа
            nums[i] = Math.random() * 100;
        sort(nums);  // Они сортируются
        for(int i = 0; i < nums.length; i++) // и распечатываются
            System.out.println(nums[i]);
        }
}
```

Вычисление простых чисел

В примере 1.15 с применением алгоритма «Решето Эратосфена» вычисляется наибольшее простое число, не превосходящее заданное значение. Этот алгоритм находит простые числа путем исключения всех чисел, кратных меньшим простым. Как и в примере 1.14, здесь нет новых синтаксических конструкций языка Java, но этой красивой и нетривиальной программой приятно закончить настоящую главу. Эта программа может показаться обманчиво простой, но в ней много чего происходит, так что убедитесь, что вы хорошо поняли, как она справляется с простыми числами.

Пример 1.15. Sieve.java

```
package com.davidflanagan.examples.basics;
/**
 * Эта программа вычисляет простые числа, применяя алгоритм
 * «Решето Эратосфена»: уберите числа, кратные меньшим простым числам,
 * и все оставшиеся будут простыми. Она печатает наибольшее число,
 * не превосходящее аргумент, заданный в командной строке.
 **/
public class Sieve {
 public static void main(String[] args) {
     // Мы вычислим все простые числа, не превосходящие заданное значение, или,
     // если аргумент не указан, все простые числа, не превосходящие 100.
     int max = 100;  // Присваиваем значение, принимаемое по умолчанию
     try { max = Integer.parseInt(args[0]); } // Анализируем заданный
                                              // пользователем аргумент
     catch (Exception e) {}  // Молча игнорируем исключения
     // Создаем массив, где для каждого числа указано, простое оно или нет.
     boolean[] isprime = new boolean[max+1];
     // Предполагаем, что все числа простые, пока не доказано обратное.
     for(int i = 0; i <= max; i++) isprime[i] = true;
     // Мы, однако, знаем, что 0 и 1 - не простые числа.
     // Обратите на это внимание.
     isprime[0] = isprime[1] = false;
     // Чтобы вычислить все простые числа меньше max, нужно убрать
     // числа, кратные всем целым, меньшим, чем квадратный корень из max.
     int n = (int) Math.ceil(Math.sqrt(max)); // См. класс java.lang.Math
     // Теперь для каждого целого i от 0 до n:
```

```
        //     Если i простое число, тогда никакое из кратных ему не простое,
        //     отметим это в нашем массиве. Если i не простое число, кратные
        //     ему уже удалены (в качестве кратных его простому делителю),
        //     значит, этот случай пропускаем.
        for(int i = 0; i <= n; i++) {
            if (isprime[i])                          // Если i - простое число,
                for(int j = 2*i; j <= max; j = j + i) // цикл по кратным,
                   isprime[j] = false;               // они не являются простыми.
        }
        // Теперь найдем наибольшее простое:
        int largest;
        for(largest = max; !isprime[largest]; largest--) ; // Пустое тело цикла
        // Выведем результат
       System.out.println("Наибольшее простое число, не превосходящее " + max +
        " это " + largest);
    }
}
```

Упражнения

1-1. Напишите программу, которая считает от 1 до 15, печатает каждое число и затем считает двойками в обратном направлении до 1, снова печатая каждое число.

1-2. Каждый элемент в ряде Фибоначчи получается сложением двух предыдущих. Какой ряд получится, если складывать три предыдущих числа? Напишите программу, печатающую первые 20 элементов такого ряда.

1-3. Напишите программу, получающую в качестве аргументов в командной строке два числа и строку и печатающую подстроку заданной строки, определенную заданными числами. Например:

`% java Substring hello 1 3` должна вывести: `ell`

Обработайте всевозможные исключения, обусловленные неправильным вводом.

1-4. Напишите программу, которая считывает в интерактивном режиме вводимые пользователем строки и печатает их в обратном порядке. Исполнение программы прекращается, когда пользователь введет «tiuq».

1-5. Класс `SortNumbers` показывает, как можно отсортировать массив чисел типа `double`. Напишите программу, применяющую этот класс для сортировки 100 чисел с плавающей точкой. Затем в интерактивном режиме предложите пользователю ввести число и отобразите соседние с ним в массиве большее и меньшее числа. Чтобы найти нужную позицию в отсортированном массиве, примените эффективный алгоритм двоичного поиска.