ГЛАВА

13 Ввод-вывод, аплеты и прочие вопросы

Эта глава посвящена двум наиболее важным пакетам в Java: `io` и `applet`. Пакет `io` поддерживает базовую систему ввода-вывода в Java, включая файловый ввод-вывод, а пакет `applet` — аплеты. Поддержка ввода-вывода и аплетов осуществляется библиотеками базового прикладного программного интерфейса (API), а не ключевыми словами языка Java. По этой причине углубленное обсуждение этих вопросов приведено в части II данной книги, где рассматриваются классы прикладного программного интерфейса Java API. А в этой главе представлены основы этих двух подсистем с целью показать, каким образом они интегрированы в язык Java и вписываются в общий контекст программирования на нем и его исполняющую среду. В главе рассматриваются также оператор `try` с ресурсами и недавно внедренные ключевые слова Java: `transient`, `volatile`, `instanceof`, `native`, `strictfp` и `assert`. И в завершение описаны статический импорт и особенности применения ключевого слова `this`, а также дано введение в компактные профили, внедренные в версии JDK 8.

Основы ввода-вывода

Читая предыдущие 12 глав этой книги, вы, вероятно, обратили внимание на то, что в приведенных до сих пор примерах программ было задействовано не так много операций ввода-вывода. По существу, никаких методов ввода-вывода, кроме `print()` и `println()`, в этих примерах не применялось. Причина этого проста: большинство реальных прикладных программ на Java не являются текстовыми консольными программами, а содержат графический пользовательский интерфейс (ГПИ), построенный на основе библиотек AWT, Swing или JavaFX для взаимодействия с пользователем, или же они являются веб-приложениями. Текстовые консольные программы отлично подходят в качестве учебных примеров, но они имеют весьма незначительное практическое применение. К тому же поддержка консольного ввода-вывода в Java ограничена и не очень удобна в употреблении — даже в простейших программах. Таким образом, текстовый консольный ввод-вывод не имеет большого практического значения для программирования на Java.

Несмотря на все сказанное выше, в Java предоставляется сильная и универсальная поддержка файлового и сетевого ввода-вывода. Система ввода-вывода в Java целостна и последовательна. Если усвоить ее основы, то овладеть всем остальным

будет очень просто. Здесь дается лишь общий обзор ввода-вывода, а подробное его описание приводится в главах 20 и 21.

Потоки ввода-вывода

В программах на Java создаются потоки ввода-вывода. *Поток ввода-вывода* — это абстракция, которая поставляет или потребляет информацию. Поток ввода-вывода связан с физическим устройством через систему ввода-вывода в Java. Все потоки ввода-вывода ведут себя одинаково, несмотря на отличия в конкретных физических устройствах, с которыми они связаны. Таким образом, одни и те же классы и методы ввода-вывода применимы к разнотипным устройствам. Это означает, что абстракция потока ввода может охватывать разные типы ввода: из файла на диске, клавиатуры или сетевого соединения. Аналогично поток вывода может обращаться к консоли, файлу на диске или сетевому соединению. Потоки ввода-вывода предоставляют ясный способ организации ввода-вывода, избавляя от необходимости разбираться в отличиях, например, клавиатуры от сети. В языке Java потоки ввода-вывода реализуются в пределах иерархии классов, определенных в пакете `java.io`.

На заметку! Помимо потокового ввода-вывода, определенного в пакете `java.io`, в Java предоставляется также буферный и канальный ввод-вывод, определенный в пакете `java.nio` и его подчиненных пакетах. Эти разновидности ввода-вывода рассматриваются в главе 21.

Потоки ввода-вывода байтов и символов

В Java определяются два вида потоков ввода-вывода: байтов и символов. *Потоки ввода-вывода байтов* предоставляют удобные средства для управления вводом и выводом отдельных байтов. Эти потоки используются, например, при чтении и записи двоичных данных. *Потоки ввода-вывода символов* предоставляют удобные средства управления вводом и выводом отдельных символов. С этой целью в них применяется кодировка в Юникоде, допускающая интернационализацию. Кроме того, потоки ввода-вывода символов оказываются порой более эффективными, чем потоки ввода-вывода байтов.

В первоначальной версии Java 1.0 потоки ввода-вывода символов отсутствовали, и поэтому весь ввод-вывод имел байтовую организацию. Потоки ввода-вывода символов были внедрены в версии Java 1.1, сделав не рекомендованным к употреблению некоторые классы и методы, поддерживавшие ввод-вывод с байтовой организацией. Устаревший код, в котором не применяются потоки ввода-вывода байтов, встречается все реже, но иногда он все еще применяется. Как правило, устаревший код приходится обновлять, если это возможно, чтобы воспользоваться преимуществами потоков ввода-вывода символов.

Следует также иметь в виду, что на самом низком уровне весь ввод-вывод по-прежнему имеет байтовую организацию. А потоки ввода-вывода символов лишь предоставляют удобные и эффективные средства для обращения с символами. В последующих разделах дается краткий обзор потоков ввода-вывода с байтовой и символьной организацией.

Классы потоков ввода-вывода байтов

Потоки ввода-вывода байтов определены в двух иерархиях классов. На вершине этих иерархий находятся абстрактные классы `InputStream` и `OutputStream`. У каждого из этих абстрактных классов имеется несколько конкретных подклассов, в которых учитываются отличия разных устройств, в числе файлов на диске, сетевых соединений и даже буферов памяти. Классы потоков ввода-вывода байтов из пакета `java.io` перечислены в табл. 13.1. Одни из этих классов описываются ниже, а другие — в части II данной книги. Не следует, однако, забывать о необходимости импортировать пакет `java.io`, чтобы воспользоваться классами потоков ввода-вывода.

Таблица 13.1. Классы потоков ввода-вывода байтов из пакета `java.io`

Класс потока ввода-вывода	Назначение
`BufferedInputStream`	Буферизированный поток ввода
`BufferedOutputStream`	Буферизированный поток вывода
`ByteArrayInputStream`	Поток ввода, читающий байты из массива
`ByteArrayOutputStream`	Поток вывода, записывающий байты в массив
`DataInputStream`	Поток ввода, содержащий методы для чтения данных стандартных типов, определенных в Java
`DataOutputStream`	Поток вывода, содержащий методы для записи данных стандартных типов, определенных в Java
`FileInputStream`	Поток ввода, читающий данные из файла
`FileOutputStream`	Поток вывода, записывающий данные в файл
`FilterInputStream`	Реализует абстрактный класс **`InputStream`**
`FilterOutputStream`	Реализует абстрактный класс **`OutputStream`**
`InputStream`	Абстрактный класс, описывающий поток ввода
`ObjectInputStream`	Поток ввода объектов
`ObjectOutputStream`	Поток вывода объектов
`OutputStream`	Абстрактный класс, описывающий поток вывода
`PipedInputStream`	Канал ввода
`PipedOutputStream`	Канал вывода
`PrintStream`	Поток вывода, содержащий методы **`print()`** и **`println()`**
`PushbackInputStream`	Поток ввода, поддерживающий возврат одного байта обратно в поток ввода
`SequenceInputStream`	Поток ввода, состоящий из двух и более потоков ввода, данные из которых читаются по очереди

В абстрактных классах `InputStream` и `OutputStream` определяется ряд ключевых методов, реализуемых в других классах потоков ввода-вывода. Наиболее важными среди них являются методы `read()` и `write()`, читающие и записыва-

ющие байты данных соответственно. Оба эти метода объявлены как абстрактные в классах `InputStream` и `OutputStream`, а в производных классах они переопределяются.

Классы потоков ввода-вывода символов

Потоки ввода-вывода символов также определены в двух иерархиях классов. На вершине этих иерархий находятся два абстрактных класса — `Reader` и `Writer`. Эти абстрактные классы управляют потоками символов в Юникоде. Для каждого из них в Java предусмотрен ряд конкретных подклассов. Классы потоков ввода-вывода символов перечислены в табл. 13.2.

Таблица 13.2. Классы потоков ввода-вывода символов из пакета `java.io`

Класс потока ввода-вывода	Назначение
`BufferedReader`	Буферизированный поток ввода символов
`BufferedWriter`	Буферизированный поток вывода символов
`CharArrayReader`	Поток ввода, читающий символы из массива
`CharArrayWriter`	Поток вывода, записывающий символы в массив
`FileReader`	Поток ввода, читающий символы из файла
`FileWriter`	Поток вывода, записывающий символы в файл
`FilterReader`	Фильтрованный поток чтения
`FilterWriter`	Фильтрованный поток записи
`InputStreamReader`	Поток ввода, преобразующий байты в символы
`LineNumberReader`	Поток ввода, подсчитывающий строки
`OutputStreamWriter`	Поток вывода, преобразующий символы в байты
`PipedReader`	Канал ввода
`PipedWriter`	Канал вывода
`PrintWriter`	Поток вывода, содержащий методы **`print()`** и **`println()`**
`PushbackReader`	Поток ввода, позволяющий возвращать символы обратно в поток ввода
`Reader`	Абстрактный класс, описывающий поток ввода символов
`StringReader`	Поток ввода, читающий символы из строки
`StringWriter`	Поток вывода, записывающий символы в строку
`Writer`	Абстрактный класс, описывающий поток вывода символов

В абстрактных классах `Reader` и `Writer` определяется ряд ключевых методов, реализуемых в других классах потоков ввода-вывода. Наиболее важными среди них являются методы `read()` и `write()`, читающие и записывающие байты данных соответственно. Оба эти метода объявлены как абстрактные в классах `Reader` и `Writer`, а в производных классах они переопределяются.

Предопределенные потоки ввода-вывода

Как вам должно быть уже известно, все программы на Java автоматически импортируют пакет `java.lang`. В этом пакете определен класс `System`, инкапсулирующий некоторые свойства исполняющей среды Java. Используя некоторые из его методов, можно, например, получить текущее время и настройки различных параметров, связанных с системой. Класс `System` содержит также три переменные предопределенных потоков ввода-вывода: `in`, `out` и `err`. Эти переменные объявлены в классе `System` как `public`, `static` и `final`. Это означает, что они могут быть использованы в любой другой части прикладной программы без обращения к конкретному объекту класса `System`.

Переменная `System.out` ссылается на стандартный поток вывода. По умолчанию это консоль. Переменная `System.in` ссылается на стандартный поток ввода, которым по умолчанию является клавиатура. А переменная `System.err` ссылается на стандартный поток вывода ошибок, которым по умолчанию также является консоль. Но эти стандартные потоки могут быть перенаправлены на любое совместимое устройство ввода-вывода.

Переменная `System.in` содержит объект типа `InputStream`, а переменные `System.out` и `System.err` содержат объекты типа `PrintStream`. Это потоки ввода-вывода байтов, хотя они, как правило, используются для чтения символов с консоли и записи символов на консоль. Как будет показано далее, их можно, если требуется, заключить в оболочки потоков ввода-вывода символов.

В примерах, приведенных в предыдущих главах, использовался стандартный поток вывода `System.out`. Аналогичным образом можно воспользоваться и стандартным потоком вывода `System.err`. Как поясняется в следующем разделе, пользоваться потоком `System.in` немного сложнее.

Чтение данных, вводимых с консоли

Организовать ввод данных с консоли в версии Java 1.0 можно было только с помощью потока ввода байтов. Такое по-прежнему возможно и теперь, но для коммерческого применения чтение данных, вводимых с консоли, предпочтительнее организовать с помощью потока ввода символов. Это значительно упрощает возможности интернационализации и сопровождения разрабатываемых программ.

В Java данные, вводимые с консоли, читаются из стандартного потока ввода `System.in`. Чтобы получить поток ввода символов, присоединив его к консоли, следует заключить стандартный поток ввода `System.in` в оболочку объекта класса `BufferedReader`, поддерживающего буферизованный поток ввода. Ниже приведен чаще всего используемый конструктор этого класса.

`BufferedReader(Reader` *`поток_чтения_вводимых_данных`*`)`

Здесь параметр *`поток_чтения_вводимых_данных`* обозначает поток, который связывается с создаваемым экземпляром класса `BufferedReader`. Класс `Reader` является абстрактным. Одним из производных от него конкретных подклассов является класс `InputStreamReader`, преобразующий байты в символы. Для получе-

ния объекта типа InputStreamReader, связанного со стандартным потоком ввода System.in, служит следующий конструктор:

InputStreamReader(InputStream *поток_ввода*)

Переменная System.in ссылается на объект класса InputStream и поэтому должна быть указана в качестве параметра *поток_ввода*. В конечном итоге получается приведенная ниже строка кода, в которой создается объект типа BufferedReader, связанный с клавиатурой. После выполнения этой строки кода переменная экземпляра br будет содержать поток ввода символов, связанный с консолью через стандартный поток ввода System.in.

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

Чтение символов

Для чтения символа из потока ввода типа BufferedReader служит метод read(). Ниже показана версия метода read(), которая будет использоваться в приведенных далее примерах программ.

int read() throws IOException

Каждый раз, когда вызывается метод read(), он читает символ из потока ввода и возвращает его в виде целочисленного значения. По достижении конца потока возвращается значение **-1**. Как видите, метод read() может сгенерировать исключение типа IOException.

В приведенном ниже примере программы демонстрируется применение метода read() для чтения символов с консоли до тех пор, пока пользователь не введет символ "q". Следует заметить, что любые исключения, возникающие при вводе-выводе, просто генерируются в методе main(). Такой подход распространен при чтении данных с консоли в простых примерах программ, аналогичных представленным в этой книге, но в более сложных прикладных программах исключения можно обрабатывать явным образом.

```
// Использовать класс BufferedReader для чтения символов с консоли
import java.io.*;

class BRRead {
    public static void main(String args[]) throws IOException
    {
        char c;
        BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));
        System.out.println("Введите символы, 'q' - для выхода.");
        // читать символы
        do {
            c = (char) br.read();
            System.out.println(c);
        } while(c != 'q');
    }
}
```

Ниже приведен пример выполнения данной программы.

```
Введите символы, 'q' - для выхода.
123abcq
1
2
3
a
b
c
q
```

Результат выполнения данной программы может выглядеть не совсем так, как предполагалось, потому что стандартный поток ввода `System.in` по умолчанию является буферизованным построчно. Это означает, что никакие вводимые данные на самом деле не передаются программе до тех пор, пока не будет нажата клавиша <Enter>. Нетрудно догадаться, что эта особенность делает метод `read()` малопригодным для организации ввода с консоли в диалоговом режиме.

Чтение символьных строк

Для чтения символьных строк с клавиатуры служит версия метода `readLine()`, который является членом класса `BufferedReader`. Его общая форма приведена ниже. Как видите, этот метод возвращает объект типа `String`.

```
String readLine() throws IOException
```

В приведенном ниже примере программы демонстрируется применение класса `BufferedReader` и метода `readLine()`. Эта программа читает и выводит текстовые строки текста до тех пор, пока не будет введено слово "стоп".

```
// Чтение символьных строк с консоли средствами класса BufferedReader
import java.io.*;

class BRReadLines {
    public static void main(String args[]) throws IOException
    {
        // создать поток ввода типа BufferedReader,
        // используя стандартный поток ввода System.in
        BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));
        String str;
        System.out.println("Введите строки текста.");
        System.out.println("Введите 'стоп' для завершения.");
        do {
            str = br.readLine();
            System.out.println(str);
        } while(!str.equals("стоп"));
    }
}
```

В следующем примере программы демонстрируется простейший текстовый редактор. С этой целью сначала создается массив объектов типа `String`, а затем читаются текстовые строки, каждая из которых сохраняется в элементе массива. Чтение производится до 100 строк или до тех пор, пока не будет введено слово "стоп". Для чтения данных с консоли применяется класс `BufferedReader`.

```
// Простейший текстовый редактор
import java.io.*;

class TinyEdit {
    public static void main(String args[]) throws IOException
    {
        // создать поток ввода типа BufferedReader,
        // используя стандартный поток ввода System.in
        BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));
        String str[] = new String[100];
        System.out.println("Введите строки текста.");
        System.out.println("Введите 'стоп' для завершения.");
        for(int i=0; i<100; i++) {
            str[i] = br.readLine();
            if(str[i].equals("стоп")) break;
        }
        System.out.println("\nСодержимое вашего файла:");
        // вывести текстовые строки
        for(int i=0; i<100; i++) {
            if(str[i].equals("стоп")) break;
            System.out.println(str[i]);
        }
    }
}
```

Ниже приведен пример выполнения данной программы.

```
Введите строки текста.
Введите 'стоп' для завершения.
Это строка один.
Это строка два.
Java упрощает работу со строками.
Просто создайте объекты типа String.
стоп
Содержимое вашего файла:
Это строка один.
Это строка два.
Java упрощает работу со строками.
Просто создайте объекты типа String.
```

Запись данных, выводимых на консоль

Вывод данных на консоль проще всего организовать с помощью упоминавшихся ранее методов `print()` и `println()`, которые применяются в большинстве примеров из этой книги. Эти методы определены в классе `PrintStream` (он является типом объекта, на который ссылается переменная `System.out`). Несмотря на то что стандартный поток `System.out` служит для вывода байтов, его можно вполне применять в простых программах для вывода данных. Тем не менее в следующем разделе описывается альтернативный ему поток вывода символов.

Класс `PrintStream` описывает поток вывода и является производным от класса `OutputStream`, поэтому в нем реализуется также низкоуровневый метод `write()`. Следовательно, метод `write()` можно применять для записи данных, выводимых на консоль. Ниже приведена простейшая форма метода `write()`, определенного в классе `PrintStream`.

void write(int *байтовое_значение*)

Этот метод записывает байт, передаваемый в качестве параметра *байтовое_значение*. Несмотря на то что параметр *байтовое_значение* объявлен как целочисленный, записываются только 8 его младших бит. Ниже приведен короткий пример, в котором метод `write()` применяется для вывода на экран буквы "A" с последующим переводом строки.

```
// Продемонстрировать применение метода System.out.write()
class WriteDemo {
    public static void main(String args[]) {
        int b;

        b = 'A';
        System.out.write(b);
        System.out.write('\n');
    }
}
```

Пользоваться методом `write()` для вывода данных на консоль приходится нечасто, хотя иногда это и удобно. Ведь намного проще применять для этой цели методы `print()` и `println()`.

Класс PrintWriter

Несмотря на то что стандартным потоком `System.out` вполне допустимо пользоваться для вывода данных на консоль, он все же подходит в большей степени для отладки или примеров программ, аналогичных представленным в данной книге. А для реальных программ рекомендуемым средством вывода данных на консоль служит поток записи, реализованный в классе `PrintWriter`, относящемся к категории символьных классов. Применение такого класса для консольного вывода упрощает интернационализацию прикладных программ.

В классе `PrintWriter` определяется несколько конструкторов. Ниже приведен один из тех конструкторов, которые применяются в рассматриваемых далее примерах.

PrintWriter(OutputStream *поток_вывода*, boolean *очистка*)

Здесь параметр *поток_вывода* обозначает объект типа `OutputStream`, а параметр *очистка* — очистку потока вывода всякий раз, когда вызывается (среди прочих) метод `println()`. Если параметр *очистка* принимает логическое значение `true`, то очистка потока вывода происходит автоматически, а иначе — вручную.

В классе `PrintWriter` поддерживаются методы `print()` и `println()`. Следовательно, их можно использовать таким же образом, как и в стандартном потоке вывода `System.out`. Если аргумент этих методов не относится к простому типу, то для объекта типа `PrintWriter` сначала вызывается метод `toString()`, а затем выводится результат.

Чтобы вывести данные на консоль, используя класс `PrintWriter`, следует указать стандартный поток `System.out` для вывода и его автоматическую очистку.

Например, в следующей строке кода создается объект типа PrintWriter, который связывается с консольным выводом:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

В приведенном ниже примере программы демонстрируется применение класса PrintWriter для управления выводом данных на консоль.

```
// Продемонстрировать применение класса PrintWriter
import java.io.*;

public class PrintWriterDemo {
    public static void main(String args[]) {
        PrintWriter pw = new PrintWriter(System.out, true);

        pw.println("Это строка");
        int i = -7;
        pw.println(i);
        double d = 4.5e-7;
        pw.println(d);
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Это строка
-7
4.5E-7
```

Напомним, что стандартный поток System.out вполне пригоден для вывода простого текста на консоль на стадии овладения языком Java или отладки прикладных программ, тогда как класс PrintWriter упрощает интернационализацию реальных программ. Но поскольку применение класса PrintWriter не дает никаких преимуществ в примерах простых программ, то для вывода данных на консоль будет и далее применяться стандартный поток System.out.

Чтение и запись данных в файлы

В Java предоставляется немало классов и методов, позволяющих читать и записывать данные в файлы. Прежде всего, следует заметить, что тема ввода-вывода данных в файлы весьма обширна и подробно обсуждается в части II данной книги. А в этом разделе будут представлены основные способы чтения и записи данных в файл. И хотя для этой цели применяются потоки ввода-вывода байтов, упоминаемые здесь способы нетрудно приспособить и под потоки ввода-вывода символов.

Для ввода-вывода данных в файлы чаще всего применяются классы FileInputStream и FileOutputStream, которые создают потоки ввода-вывода байтов, связанные с файлами. Чтобы открыть файл для ввода-вывода данных, достаточно создать объект одного из этих классов, указав имя файла в качестве аргумента конструктора. У обоих классов имеются и дополнительные конструкторы, но в представленных далее примерах будут употребляться только следующие конструкторы:

```
FileInputStream(String имя_файла) throws FileNotFoundException
FileOutputStream(String имя_файла) throws FileNotFoundException
```

где параметр *имя_файла* обозначает имя того файла, который требуется открыть. Если при создании потока ввода файл не существует, то генерируется исключение типа `FileNotFoundException`. А если при создании потока вывода файл нельзя открыть или создать, то и в этом случае генерируется исключение типа `FileNotFoundException`. Класс исключения `FileNotFoundException` является производным от класса `IOException`. Когда файл открыт для вывода, любой файл, существовавший ранее под тем же самым именем, уничтожается.

На заметку! В тех случаях, когда присутствует диспетчер защиты, некоторые файловые классы, в том числе `FileInputStream` и `FileOutputStream`, генерируют исключение типа `SecurityException`, если при попытке открыть файл обнаруживается нарушение защиты. По умолчанию в прикладных программах, запускаемых на выполнение по команде **`java`**, диспетчер защиты не применяется. Поэтому в примерах, демонстрирующих организацию ввода-вывода в данной книге, вероятность генерирования исключения типа `SecurityException` не отслеживается. Но в других видах прикладных программ (например, аплетах) диспетчер защиты обычно применяется, и поэтому операции ввода-вывода данных в файлы вполне могут привести в них к исключению типа `SecurityException`. В таком случае следует организовать соответствующую обработку этого исключения.

Завершив работу с файлом, его нужно закрыть. Для этой цели служит метод `close()`, реализованный в классах `FileInputStream` и `FileOutputStream`:

```
void close() throws IOException
```

Закрытие файла высвобождает выделенные для него системные ресурсы, позволяя использовать их для других файлов. Неудачный исход закрытия файла может привести к "утечкам памяти", поскольку неиспользуемые ресурсы оперативной памяти останутся выделенными.

На заметку! Начиная с версии JDK 7 метод `close()` определяется в интерфейсе `AutoCloseable` из пакета `java.lang`. Интерфейс `AutoCloseable` наследует от интерфейса `Closeable` из пакета `java.io`. Оба интерфейса реализуются классами потоков ввода-вывода, включая классы `FileInputStream` и `FileOutputStream`.

Следует заметить, что имеются два основных способа закрытия файла, когда он больше не нужен. При первом, традиционном способе метод `close()` вызывается явно, когда файл больше не нужен. Именно такой способ применялся во всех версиях Java до JDK 7, и поэтому он присутствует во всем коде, написанном до версии JDK 7. А при втором способе применяется оператор `try` с ресурсами, внедренный в версии JDK 7. Этот оператор автоматически закрывает файл, когда он больше не нужен. И в этом случае отпадает потребность в явных вызовах метода `close()`. Но поскольку существует немалый объем кода, который написан до версии JDK 7 и все еще эксплуатируется и сопровождается, то по-прежнему важно знать и владеть традиционным способом закрытия файлов. Поэтому сначала будет рассмотрен именно этот способ, а новый, автоматизированный способ описывается в следующем разделе.

Чтобы прочитать данные из файла, можно воспользоваться версией метода `read()`, определенной в классе `FileInputStream`. Та его версия, которая применяется в представленных далее примерах, выглядит следующим образом:

```
int read() throws IOException
```

Всякий раз, когда вызывается метод `read()`, он выполняет чтение одного байта из файла и возвращает его в виде целочисленного значения. А если достигнут конец файла, то возвращается значение **-1**. Этот метод может сгенерировать исключение типа `IOException`.

В приведенном ниже примере программы метод `read()` применяется для ввода из файла, содержащего текст в коде ASCII, который затем выводится на экран. Имя файла указывается в качестве аргумента командной строки.

```
/* Отображение содержимого текстового файла.
   Чтобы воспользоваться этой программой, укажите
   имя файла, который требуется просмотреть.
   Например, чтобы просмотреть файл TEST.TXT,
   введите в командной строке следующую команду:

   java ShowFile TEST.TXT
*/

import java.io.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;
        FileInputStream fin;

        // сначала убедиться, что имя файла указано
        if(args.length != 1) {
            System.out.println("Использование: ShowFile имя_файла");
            return;
        }

        // Попытка открыть файл
        try {
            fin = new FileInputStream(args[0]);
        } catch(FileNotFoundException e) {
            System.out.println("Невозможно открыть файл");
            return;
        }

        // Теперь файл открыт и готов к чтению.
        // Далее из него читаются символы до тех пор,
        // пока не встретится признак конца файла
        try {
            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);
        } catch(IOException e) {
            System.out.println("Ошибка чтения из файла");
        }

        // закрыть файл
        try {
```

```
            fin.close();
        } catch(IOException e) {
            System.out.println("Ошибка закрытия файла");
        }
    }
}
```

Обратите внимание в данном примере программы на блок операторов `try/catch`, обрабатывающий ошибки, которые могут произойти при вводе-выводе. Каждая операция ввода-вывода проверяется на наличие исключений, и если исключение возникает, то оно обрабатывается. В простых программах или примерах кода исключения, возникающие в операциях ввода-вывода, как правило, генерируются в методе `main()`, как это делалось в предыдущих примерах ввода-вывода на консоль. Кроме того, в реальном прикладном коде иногда оказывается полезно, чтобы исключение распространялось в вызывающую часть программы, уведомляя ее о неудачном исходе операции ввода-вывода. Но ради демонстрации в большинстве примеров файлового ввода-вывода, представленных в данной книге, все исключения, возникающие в операциях ввода-вывода, обрабатываются явным образом.

В приведенном выше примере поток ввода закрывается после чтения из файла, но имеется и другая возможность, которая нередко оказывается удобной. Она подразумевает вызов метода `close()` из блока оператора `finally`. В таком случае все методы, получающие доступ к файлу, содержатся в блоке оператора `try`, тогда как блок оператора `finally` служит для закрытия файла. Таким образом, файл будет закрыт независимо от того, как завершится блок оператора `try`. Если обратиться к приведенному выше примеру, то в свете только что сказанного блок оператора `try`, где читаются данные из файла, может быть переписан следующим образом:

```
try {
    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);
} catch(IOException e) {
    System.out.println("Ошибка чтения из файла");
} finally {
    // закрыть файл при выходе из блока оператора try
    try {
        fin.close();
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла");
    }
}
```

В данном случае это не так важно. Тем не менее одно из преимуществ такого подхода состоит в том, что если выполнение кода, в котором происходит обращение к файлу, прекращается из-за каких-нибудь исключений, не связанных с операциями ввода-вывода, то файл все равно будет закрыт в блоке оператора `finally`.

Иногда проще заключить все части программы, открывающие файл и получающие доступ к его содержимому, в один блок оператора `try`, вместо того чтобы разделять его на два блока, а затем закрыть файл в блоке оператора `finally`. В качестве иллюстрации ниже показан другой способ написания программы `ShowFile` из предыдущего примера.

```
/* Отображение содержимого текстового файла.
   Чтобы воспользоваться этой программой, укажите
   имя файла, который требуется просмотреть.
   Например, чтобы просмотреть файл TEST.TXT,
   введите в командной строке следующую команду:

   java ShowFile TEST.TXT

   В этом варианте программы код, открывающий и получающий
   доступ к файлу, заключен в один блок оператора try.
   Файл закрывается в блоке оператора finally.
*/

import java.io.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;
        FileInputStream fin = null;

        // сначала убедиться, что имя файла указано
        if(args.length != 1) {
            System.out.println("Использование: ShowFile имя_файла");
            return;
        }

        // В следующем коде сначала открывается файл, а затем
        // из него читаются символы до тех пор, пока не встретится
        // признак конца файла
        try {
            fin = new FileInputStream(args[0]);

            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(FileNotFoundException e) {
            System.out.println("Файл не найден.");
        } catch(IOException e) {
            System.out.println("Произошла ошибка ввода-вывода");
        } finally {
            // закрыть файл в любом случае
            try {
                if(fin != null) fin.close();
            } catch(IOException e) {
                System.out.println("Ошибка закрытия файла");
            }
        }
    }
}
```

Как видите, при таком подходе объект `fin` инициализируется пустым значением `null`. А в блоке оператора `finally` файл закрывается только в том случае, если объект `fin` не содержит пустое значение `null`. Такой подход оказывается работоспособным потому, что объект `fin` не будет содержать пустое значение `null` только том случае, если файл успешно открыт. Таким образом, метод `close()` не вызывается, если при открытии файла возникает исключение.

Последовательность операторов try/catch в приведенном выше примере можно сделать более краткой. Класс исключения FileNotFoundException является производным от класса IOException, и поэтому обрабатывать отдельно его исключение совсем не обязательно. В качестве примера ниже приведена переделанная последовательность операторов try/catch без перехвата исключения типа FileNotFoundException. В данном случае отображается стандартное сообщение об исключительной ситуации, описывающее возникшую ошибку.

```
try {
    fin = new FileInputStream(args[0]);

    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);

} catch(IOException e) {
    System.out.println("Ошибка ввода-вывода: " + e);
} finally {
    // закрыть файл в любом случае
    try {
        if(fin != null) fin.close();
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла");
    }
}
```

При таком подходе любая ошибка, в том числе и ошибка открытия файла, обрабатывается одним оператором catch. Благодаря своей краткости такой подход употребляется в большинстве примеров организации ввода-вывода, представленных в данной книге. Но этот подход не годится в тех случаях, когда требуется иначе отреагировать на неудачный исход открытия файла, например, когда пользователь может неправильно ввести имя файла. В таком случае можно было бы, например, запросить правильное имя файла, прежде чем переходить к блоку оператора try, где происходит обращение к файлу.

Для записи в файл можно воспользоваться методом write(), определенным в классе FileOutputStream. В своей простейшей форме этот метод выглядит следующим образом:

```
void write(int байтовое_значение) throws IOException
```

Этот метод записывает в файл байт, переданный ему в качестве параметра *байтовое_значение*. Несмотря на то что параметр *байтовое_значение* объявлен как целочисленный, в файл записываются только его младшие восемь бит. Если при записи возникает ошибка, генерируется исключение типа IOException. В следующем примере программы метод write() применяется для копирования файла:

```
/* Копирование файла.
   Чтобы воспользоваться этой программой, укажите
   имена исходного и целевого файлов.
   Например, чтобы скопировать файл FIRST.TXT в файл SECOND.TXT,
   введите в командной строке следующую команду:

   java CopyFile FIRST.TXT SECOND.TXT
*/
```

```
import java.io.*;

class CopyFile {
    public static void main(String args[]) throws IOException
    {
        int i;
        FileInputStream fin = null;
        FileOutputStream fout = null;

        // сначала убедиться, что указаны имена обоих файлов
        if(args.length != 2) {
            System.out.println("Использование: CopyFile откуда куда");
            return;
        }

        // копировать файл
        try {
            // попытаться открыть файлы
            fin = new FileInputStream(args[0]);
            fout = new FileOutputStream(args[1]);

            do {
                i = fin.read();
                if(i != -1) fout.write(i);
            } while(i != -1);

        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        } finally {
            try {
                if(fin != null) fin.close();
            } catch(IOException e2) {
                System.out.println("Ошибка закрытия файла ввода");
            }
            try {
                if(fout != null) fout.close();
            } catch(IOException e2) {
                System.out.println("Ошибка закрытия файла вывода");
            }
        }
    }
}
```

Обратите внимание на то, что в данном примере программы при закрытии файлов используются два отдельных блока оператора `try`. Этим гарантируется, что оба файла будут закрыты, даже если при вызове метода `fin.close()` будет сгенерировано исключение. Обратите также внимание на то, что все потенциальные ошибки ввода-вывода обрабатываются в двух приведенных выше программах с помощью исключений, в отличие от других языков программирования, где для уведомления о файловых ошибках используются коды ошибок. Исключения в Java не только упрощают обращение с файлами, но и позволяют легко отличать условие достижения конца файла от файловых ошибок во время ввода.

Автоматическое закрытие файла

В примерах программ из предыдущего раздела метод `close()` вызывался явно, чтобы закрыть файл, как только он окажется ненужным. Как упоминалось ранее,

такой способ закрытия файлов применялся до версии JDK 7. И хотя он все еще допустим и применим, в версии JDK 7 появилась новая возможность, предлагающая иной способ управления такими ресурсами, как потоки ввода-вывода в файлы: автоматическое завершение процесса. Эту возможность иногда еще называют *автоматическим управлением ресурсами* (ARM), и основывается она на усовершенствованной версии оператора `try`. Главное преимущество автоматического управления ресурсами заключается в предотвращении ситуаций, когда файл (или другой ресурс) не освобождается по невнимательности, если он больше не нужен. Как пояснялось ранее, если забыть по какой-нибудь причине закрыть файл, это может привести к утечке памяти и другим осложнениям.

Как упоминалось выше, автоматическое управление ресурсами основывается на усовершенствованной форме оператора `try`. Ниже приведена его общая форма.

```
try (спецификация_ресурса) {
    // использование ресурса
}
```

Здесь *спецификация_ресурса* обозначает оператор, объявляющий и инициализирующий такой ресурс, как поток ввода-вывода в файл. Он состоит из объявления переменной, где переменная инициализируется ссылкой на управляемый объект. По завершении блока оператора `try` ресурс автоматически освобождается. Для файла это означает, что он автоматически закрывается, а следовательно, отпадает необходимость вызывать метод `close()` явно. Безусловно, эта новая форма оператора `try` может также включать в себя операторы `finally` и `catch`. Она называется оператором `try` *с ресурсами*.

Оператор `try` с ресурсами применяется только с теми ресурсами, которые реализуют интерфейс `AutoCloseable`, определенный в пакете `java.lang`. В этом интерфейсе определяется метод `close()`. Интерфейс `AutoCloseable` наследуется интерфейсом `Closeable` из пакета `java.io`. Оба интерфейса реализуются классами потоков ввода-вывода. Таким образом, оператор `try` с ресурсами может быть использован при обращении с потоками ввода-вывода, включая и потоки ввода-вывода в файлы. В качестве первого примера автоматического закрытия файла рассмотрим переделанную версию представленной ранее программы `ShowFile`.

```
/* В этой версии программы ShowFile оператор try с ресурсами
   применяется для автоматического закрытия файла

   Примечание: для выполнения этого кода требуется версия JDK 7
*/

import java.io.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;

        // сначала убедиться, что имя файла указано
        if(args.length != 1) {
            System.out.println("Использование: ShowFile имя_файла");
            return;
        }
```

```
        // Ниже оператор try с ресурсами применяется
        // сначала для открытия, а затем для автоматического
        // закрытия файла по завершении блока этого оператора
        try(FileInputStream fin = new FileInputStream(args[0])) {

            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(FileNotFoundException e) {
            System.out.println("Файл не найден.");
        } catch(IOException e) {
            System.out.println("Произошла ошибка ввода-вывода");
        }
    }
}
```

В приведенном выше примере программы обратите особое внимание на открытие файла в блоке оператора `try`:

```
try(FileInputStream fin = new FileInputStream(args[0])) {
```

Как видите, в той части оператора `try`, где указывается спецификация ресурса, объявляется экземпляр `fin` класса `FileInputStream`, которому затем присваивается ссылка на файл, открытый его конструктором. Таким образом, в данной версии программы переменная `fin` является локальной по отношению к блоку оператора `try`, в начале которого она создается. По завершении блока `try` поток ввода-вывода, связанный с переменной `fin`, автоматически закрывается в результате неявного вызова метода `close()`. Этот метод не нужно вызывать явно, а следовательно, исключается возможность просто забыть закрыть файл. В этом и состоит главное преимущество применения оператора `try` с ресурсами.

Однако ресурс, объявляемый в операторе `try`, неявно считается завершенным. Это означает, что присвоить ресурс после того, как он был создан, нельзя. Кроме того, область действия ресурса ограничивается пределами оператора `try` с ресурсами.

В одном операторе `try` можно организовать управление несколькими ресурсами. Для этого достаточно указать спецификацию каждого ресурса через точку с запятой. Примером тому служит приведенная ниже версия программы `CopyFile`, переделанная таким образом, чтобы использовать один оператор `try` с ресурсами для управления переменными `fin` и `fout`.

```
/* Версия программы CopyFile, использующая оператор try сресурсами.
   Она демонстрирует управление двумя ресурсами (в данном случае —
   файлами) в одном операторе try
*/

import java.io.*;

class CopyFile {
    public static void main(String args[]) throws IOException
    {
        int i;

        // сначала убедиться, что заданы оба файла
        if(args.length != 2) {
            System.out.println("Использование: CopyFile откуда куда");
```

```
            return;
        }

        // открыть два файла и управлять ними в операторе try
        try (FileInputStream fin = new FileInputStream(args[0]);
             FileOutputStream fout = new FileOutputStream(args[1]))
        {

            do {
                i = fin.read();
                if(i != -1) fout.write(i);
            } while(i != -1);

        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }
    }
}
```

Обратите внимание на то, как файлы ввода и вывода открываются в блоке оператора `try`.

```
try (FileInputStream fin = new FileInputStream(args[0]);
     FileOutputStream fout = new FileOutputStream(args[1]))
{
    // ...
```

По завершении этого блока оператора `try` будут закрыты оба ресурса в переменных `fin` и `fout`. Если сравнить эту версию программы с предыдущей, то можно заметить, что она значительно короче. Возможность упростить исходный код является дополнительным преимуществом автоматического управления ресурсами.

У оператора `try` с ресурсами имеется еще одна особенность, о которой стоит упомянуть. В общем, когда выполняется блок оператора `try`, существует вероятность того, что исключение, возникающее в блоке оператора `try`, приведет к другому исключению, которое произойдет в тот момент, когда ресурс закрывается в блоке оператора `finally`. Если это "обычный" оператор `try`, то первоначальное исключение теряется, будучи вытесненным вторым исключением. А если используется оператор `try` с ресурсами, то второе исключение *подавляется*, но не теряется. Вместо этого оно добавляется в список подавленных исключений, связанных с первым исключением. Доступ к списку подавленных исключений может быть получен с помощью метода `getSuppressed()`, определенного в классе `Throwable`.

Благодаря упомянутым выше преимуществам оператор `try` с ресурсами применяется во многих, но не во всех примерах программ, представленных в данной книге. В некоторых примерах по-прежнему применяется традиционный способ закрытия ресурсов. И на то есть несколько причин. Во-первых, существует немалый объем широко распространенного и эксплуатируемого кода, в котором применяется традиционный способ и с которым хорошо знакомы все программирующие на Java. Во-вторых, не все разрабатываемые проекты будут немедленно переведены на новую версию JDK. Некоторые программисты, вероятно, какое-то время продолжат работать в среде, предшествующей версии JDK 7, где рассматриваемая здесь улучшенная форма оператора `try` недоступна. И наконец, возможны случаи, когда явное закрытие ресурса вручную оказывается лучше, чем автоматическое. По этим причинам в некоторых примерах из этой книги будет и далее применять-

ся традиционный способ явного вызова метода `close()` для закрытия ресурсов вручную. Помимо того, что эти примеры программ демонстрируют традиционный способ управления ресурсами, они могут быть откомпилированы и запущены всеми читателями данной книги во всех системах, которыми они пользуются.

Помните! Для целей демонстрации в некоторых примерах программ из данной книги намеренно применяется традиционный способ закрытия файлов, широко распространенный в унаследованном коде. Но при написании нового кода рекомендуется применять новый способ автоматического управления ресурсами, который поддерживается в только что описанном операторе `try` с ресурсами.

Основы создания аплетов

Во всех предыдущих примерах демонстрировались программы, относящиеся к категории консольных прикладных программ на Java. Но это лишь одна из категорий прикладных программ на Java. К другой категории относятся аплеты. Как упоминалось в главе 1, *аплет* – это небольшая прикладная программа, находящаяся на веб-сервере, откуда она загружается, автоматически устанавливается и запускается как составная часть веб-документа. Как только аплет появится у клиента, он получает ограниченный доступ к ресурсам, чтобы предоставить графический пользовательский интерфейс (ГПИ) и выполнить различные вычисления, не подвергая клиента риску вирусной атаки или нарушения целостности его данных.

Начнем рассмотрение аплетов с простейшего примера, приведенного ниже.

```
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Простейший аплет", 20, 20);
    }
}
```

Этот аплет начинается с двух операторов `import`. Первый из них импортирует классы библиотеки Abstract Window Toolkit (AWT). Аплеты взаимодействуют с пользователем через библиотеку AWT, а не через классы консольного ввода-вывода. Библиотека AWT поддерживает элементарный оконный ГПИ и служит здесь в качестве введения в программирование аплетов. Нетрудно догадаться, что библиотека AWT значительно крупнее и сложнее, и полное обсуждение ее возможностей занимает несколько глав в части II данной книги. Правда, приведенный выше простой аплет весьма ограниченно использует возможности библиотеки AWT. (Для поддержки ГПИ в аплетах можно также пользоваться библиотекой Swing, но такой подход рассматривается далее в книге.) Второй оператор `import` импортирует пакет `applet`, в котором находится класс `Applet`. Каждый создаваемый аплет должен быть подклассом, прямо или косвенно производным от класса `Applet`.

В следующей строке кода из рассматриваемого здесь примера простейшего аплета объявляется класс `SimpleApplet`. Этот класс должен быть объявлен открытым (`public`), чтобы быть доступным для кода за пределами аплета.

В классе SimpleApplet объявляется метод paint(), который определен в библиотеке AWT и должен быть переопределен в аплете. Метод paint() вызывается всякий раз, когда аплет должен перерисовать результат, выводимый графическим способом. Такая ситуация может возникнуть по ряду причин. Например, окно, в котором запущен аплет, может быть перекрыто другим окном, а затем вновь открыто или же свернуто, а затем развернуто. Метод paint() вызывается в том случае, когда аплет начинает свое выполнение. Независимо от конкретной причины, всякий раз, когда аплет должен перерисовать выводимый графическим способом результат, вызывается метод paint(), принимающий единственный параметр типа Graphics. Этот параметр содержит графический контекст, описывающий графическую среду, в которой действует аплет. Графический контекст используется всякий раз, когда из аплета требуется вывести результат.

В методе paint() вызывается метод drawString() из класса Graphics. Этот метод выводит строку на позиции, задаваемой координатами X,Y. Он имеет следующую общую форму:

```
void drawString(String сообщение, int x, int y)
```

где параметр *сообщение* обозначает символьную строку, которая должна быть выведена начиная с позиции, определяемой координатами *x*,*y*. В Java верхний левый угол окна имеет координаты **0,0**. В результате вызова метода drawString() в аплете выводится символьная строка "Простейший аплет", начиная с позиции, определяемой координатами **20,20**.

Обратите внимание на то, что в аплете отсутствует метод main(). В отличие от обычных программ на Java, выполнение аплета не начинается с метода main(). На самом деле этого метода нет у большинства аплетов. Вместо этого выполнение аплета начинается, когда имя его класса передается средству просмотра аплетов или сетевому браузеру.

После ввода исходного кода аплета SimpleApplet его компиляция выполняется таким же образом, как и обычных программ. Но запуск аплета SimpleApplet осуществляется иначе. По существу, запустить аплет можно двумя способами:

- в совместимом с Java браузере;
- средством просмотра аплетов, например, стандартным инструментальным средством **appletviewer**. Это средство выполняет аплет в своем окне. Обычно это самый быстрый и простой способ проверки работоспособности аплета.

Каждый из этих способов запуска аплетов подробно описывается ниже.

Для того чтобы выполнить аплет в веб-браузере, достаточно, в частности, создать короткий HTML-файл, который должен содержать соответствующий дескриптор. В настоящее время компания Oracle рекомендует использовать для этой цели дескриптор APPLET, хотя может быть также использован дескриптор OBJECT. Подробнее о методиках развертывания аплетов речь пойдет в главе 23. В качестве примера ниже приведен HTML-файл с дескриптором APPLET для запуска аплета SimpleApplet.

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

Атрибуты `width` и `height` разметки этого HTML-файл обозначают размеры области отображения, используемой аплетом. (У дескриптора `APPLET` имеются и другие параметры, более подробно рассматриваемые в части II.) Создав рассматриваемый здесь HTML-файл, следует запустить браузер, а затем загрузить в нем этот файл, в результате чего и будет выполнен аплет `SimpleApplet`.

На заметку! Начиная с обновления 21 версии Java 7, аплеты в Java должны подписываться во избежание предупреждений о нарушении защиты при их выполнении в браузере, хотя иногда требуется предотвратить выполнение аплета. К этим переменам особенно чувствительны аплеты, хранящиеся в локальной файловой системе, в том числе и те, что получаются в результате компиляции примеров из данной книги. Так, для выполнения локального аплета в браузере, скорее всего, придется настроить параметры защиты на панели управления Java (Java Control Panel). На момент написания этой книги в компании Oracle рекомендовали пользоваться не локальными аплетами, а теми, что выполняются через веб-сервер. А в будущем ожидается, что выполнение неподписанных локальных аплетов будет блокироваться. В конечном итоге подписание аплетов, распространяемых через Интернет, станет обязательным. И хотя принципы и методики подписания аплетов (и других видов прикладных программ на Java) выходят за рамки рассмотрения этой книги, на веб-сайте компании Oracle можно найти немало подробных сведений по данному вопросу. И, наконец, чтобы опробовать представленные здесь примеры аплетов, проще всего воспользоваться упоминавшимся ранее средством просмотра аплетов **`appletviewer`**.

Чтобы опробовать аплет `SimpleApplet` в средстве просмотра аплетов, следует также выполнить приведенный выше HTML-файл. Так, если этот HTML-файл называется `RunApp.html`, то для его запуска на выполнение достаточно ввести в командной строке следующую команду:

```
C:\>appletviewer RunApp.html
```

Но существует более удобный способ, ускоряющий проверку аплетов. Для этого достаточно ввести в начале исходного файла аплета комментарий, содержащий дескриптор `APPLET`. В итоге исходный код аплета будет документирован прототипом необходимых операторов HTML, и скомпилированный аплет можно будет проверить, запуская средство просмотра аплетов вместе с исходным файлом аплета. Если применяется именно такой способ проверки аплетов, то исходный файл аплета `SimpleApplet` должен выглядеть следующим образом:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Простейший аплет", 20, 20);
    }
}
```

Подобным способом можно очень быстро проходить стадии разработки аплетов, если выполнить следующие действия.

- Отредактировать файл исходного кода аплета.
- Откомпилировать исходный код аплета.
- Запустить средство просмотра аплетов вместе исходным файлом аплета. Средство просмотра аплетов обнаружит дескриптор `APPLET` в комментарии к исходному коду аплета и запустит его на выполнение.

Окно аплета `SimpleApplet`, отображаемое средством просмотра аплетов, приведено на рис. 13.1. Разумеется, фрейм средства просмотра аплетов может отличаться своим внешним видом в зависимости от конкретной исполняющей среды. Поэтому моментальные снимки экранов, приведенные в данной книге, отражают разные исполняющие среды.

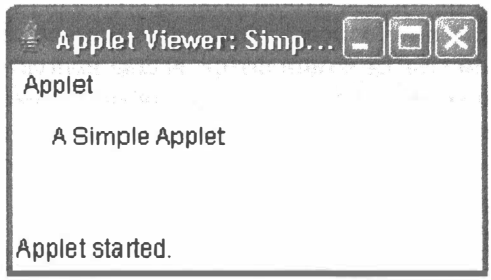

Рис. 13.1. Окно аплета `SimpleApplet`, отображаемое средством просмотра аплетов

Вопросы создания аплетов будут обсуждаться далее в данной книге, а до тех пор укажем главные их особенности, о которых нужно знать теперь.

- Аплеты не нуждаются в методе `main()`.
- Аплеты должны запускаться из средства просмотра аплетов или совместимого с Java веб-браузера.
- Пользовательский ввод-вывод в аплетах не организуется с помощью классов потоков ввода-вывода. Вместо этого в аплетах применяется ГПИ, предоставляемый соответствующими библиотеками.

Модификаторы доступа `transient` и `volatile`

В языке Java определяются два интересных модификатора доступа: `transient` и `volatile`. Эти модификаторы предназначены для особых случаев. Когда переменная-экземпляр объявлена как `transient`, ее значение не должно сохраняться, когда сохраняется объект:

```
class T {
    transient int a; // не сохранится
    int b;           // сохранится
}
```

Если в данном примере кода объект типа T записывается в область постоянного хранения, то содержимое переменной a не должно сохраняться, тогда как содержимое переменной b должно быть сохранено.

Модификатор доступа volatile сообщает компилятору, что модифицируемая им переменная может быть неожиданно изменена в других частях программы. Одна из таких ситуаций возникает в многопоточных программах, где иногда у двух или более потоков исполнения имеется совместный доступ к одной и той же переменной. Из соображений эффективности, в каждом потоке может храниться своя закрытая копия этой переменной. Настоящая (или *главная*) копия переменной обновляется в разные моменты, например, при входе в синхронизированный метод. Такой подход вполне работоспособен, но не всегда оказывается достаточно эффективным. Иногда требуется, чтобы главная копия переменной постоянно отражала ее текущее состояние. И для этого достаточно объявить переменную как volatile, сообщив тем самым компилятору всегда использовать главную копию этой переменной (или хотя бы поддерживать любые закрытые ее копии обновляемыми по главной копии, и наоборот). Кроме того, доступ к главной копии переменной должен осуществляться в том же порядке, что и к любой закрытой копии.

Применение оператора instanceof

Иногда тип объекта полезно выяснить во время выполнения. Например, в одном потоке исполнения объекты разных типов могут формироваться, а в другом потоке исполнения — использоваться. В таком случае удобно выяснить тип каждого объекта, получаемого в обрабатывающем потоке исполнения. Тип объекта во время выполнения не менее важно выяснить и в том случае, когда требуется приведение типов. В Java неправильное приведение типов влечет за собой появление ошибки во время выполнения. Большинство ошибок приведения типов может быть выявлено на стадии компиляции. Но приведение типов в пределах иерархии классов может стать причиной ошибок, которые обнаруживаются только во время выполнения. Например, суперкласс A может порождать два подкласса: B и C. Следовательно, приведение объекта класса B или C к типу A вполне допустимо, но приведение объекта класса B к типу C (и наоборот) — неверно. А поскольку объект типа A может ссылаться на объекты типа B и C, то как во время выполнения узнать, на какой именно тип делается ссылка перед тем, как выполнить приведение к типу C? Это может быть объект типа A, B или C. Если это объект типа B, то во время выполнения будет сгенерировано исключение. Для разрешения этого вопроса в Java предоставляется оператор времени выполнения instanceof, который имеет следующую общую форму:

ссылка_на_объект* instanceof *тип

где *ссылка_на_объект* обозначает ссылку на экземпляр класса, а *тип* — конкретный тип этого класса. Если *ссылка_на_объект* относится к указанному типу или может быть приведена к нему, то вычисление оператора instanceof дает в итоге логическое значение true, а иначе — логическое значение false. Таким образом, оператор instanceof — это средство, с помощью которого программа может по-

лучить сведения об объекте во время выполнения. В следующем примере программы демонстрируется применение оператора `instanceof`:

```
// Продемонстрировать применение оператора instanceof
class A {
    int i, j;
}

class B {
    int i, j;
}

class C extends A {
    int k;
}

class D extends A {
    int k;
}

class InstanceOf {
    public static void main(String args[]) {
        A a = new A();
        B b = new B();
        C c = new C();
        D d = new D();
        if(a instanceof A)
            System.out.println("a является экземпляром A");
        if(b instanceof B)
            System.out.println("b является экземпляром B");
        if(c instanceof C)
            System.out.println("c является экземпляром C");
        if(c instanceof A)
            System.out.println("c можно привести к типу A");

        if(a instanceof C)
            System.out.println("a можно привести к типу C");

        System.out.println();

        // сравнить с порожденными типами
        A ob;

        ob = d; // ссылка на объект d
        System.out.println("ob теперь ссылается на d");
        if(ob instanceof D)
            System.out.println("ob является экземпляром D");

        System.out.println();

        ob = c; // ссылка на c
        System.out.println("ob теперь ссылается на c");

        if(ob instanceof D)
            System.out.println("ob можно привести к типу D");
        else
            System.out.println("ob нельзя привести к типу D");

        if(ob instanceof A)
            System.out.println("ob можно привести к типу A");

        System.out.println();
```

```
        // все объекты могут быть приведены к типу Object
        if(a instanceof Object)
            System.out.println("a можно привести к типу Object");
        if(b instanceof Object)
            System.out.println("b можно привести к типу Object");
        if(c instanceof Object)
            System.out.println("c можно привести к типу к Object");
        if(d instanceof Object)
            System.out.println("d можно привести к типу к Object");
    }
}
```

Эта программа выводит следующий результат:

```
a является экземпляром A
b является экземпляром B
c является экземпляром C
c можно привести к типу A

ob теперь ссылается на d
ob является экземпляром D

ob теперь ссылается на c
ob нельзя привести к типу D
ob можно привести к типу A

a можно привести к типу Object
b можно привести к типу Object
c можно привести к типу Object
d можно привести к типу Object
```

Большинство программ не нуждается в операторе `instanceof`, поскольку типы объектов обычно известны заранее. Но этот оператор может пригодиться при разработке обобщенных процедур, оперирующих объектами из сложной иерархии классов.

Модификатор доступа `strictfp`

С появлением версии Java 2 модель вычислений с плавающей точкой стала чуть менее строгой. В частности, эта модель не требует теперь округления некоторых промежуточных результатов вычислений. В ряде случаев это предотвращает переполнение. Объявляя класс, метод или интерфейс с модификатором доступа `strictfp`, можно гарантировать, что вычисления с плавающей точкой будут выполняться таким же образом, как и в первых версиях Java. Если класс объявляется с модификатором доступа `strictfp`, все его методы автоматически модифицируются как `strictfp`.

Например, в приведенной ниже строке кода компилятору Java сообщается, что во всех методах, определенных в классе `MyClass`, следует использовать исходную модель вычислений с плавающей точкой. Откровенно говоря, большинству программистов вряд ли понадобится модификатор доступа `strictfp`, поскольку он касается лишь небольшой категории задач.

```
strictfp class MyClass { //...
```

Платформенно-ориентированные методы

Иногда, хотя и редко, возникает потребность вызвать подпрограмму, написанную на другом языке, а не на Java. Как правило, такая подпрограмма существует в виде исполняемого кода для ЦП и той среды, в которой приходится работать, т.е. в виде платформенно-ориентированного кода. Такую подпрограмму, возможно, потребуется вызвать для повышения скорости выполнения. С другой стороны, может возникнуть потребность работать со специализированной сторонней библиотекой, например, с пакетом статистических расчетов. Но поскольку программы на Java компилируются в байт-код, который затем интерпретируется (или динамически компилируется во время выполнения) исполняющей системой Java, то вызвать подпрограмму в платформенно-ориентированном коде из программы на Java, на первый взгляд, невозможно. К счастью, этот вывод оказывается ложным. Для объявления платформенно-ориентированных методов в Java предусмотрено ключевое слово `native`. Однажды объявленные как `native`, эти методы могут быть вызваны из прикладной программы на Java таким же образом, как и вызывается любой другой метод.

Чтобы объявить платформенно-ориентированный метод, его имя следует предварить модификатором доступа `native`, но не определять тело метода, как показано ниже.

```
public native int meth() ;
```

Объявив платформенно-ориентированный метод, нужно написать его и предпринять ряд относительно сложных шагов, чтобы связать его с кодом Java. Большинство платформенно-ориентированных методов пишутся на C. Механизм интеграции кода на C и программы на Java называется *интерфейсом JNI* (Java Native Interface). Подробное описание интерфейса JNI выходит за рамки данной книги, но предложенное ниже краткое описание дает достаточное представление для простого применения этого интерфейса.

На заметку! Конкретные действия, которые следует предпринять, зависят от применяемой среды Java, а также от языка, используемого для реализации платформенно-ориентированных методов. Приведенный ниже пример опирается на среду Windows для реализации платформенно-ориентированного метода на языке C. Кроме того, в рассматриваемом здесь примере применяется динамически подключаемая библиотека, но в версии JDK 8 появилась возможность создавать и статически подключаемую библиотеку.

Самый простой способ понять процесс — рассмотреть его на конкретном примере. Ниже представлена короткая программа, в которой используется платформенно-ориентированный метод `test()`. Можете ввести ее исходный текст.

```
// Простой пример применения платформенно-ориентированного метода
public class NativeDemo {
    int i;
    public static void main(String args[]) {
        NativeDemo ob = new NativeDemo();

        ob.i = 10;
```

```
        System.out.println("Содержимое переменной ob.i перед вызовом
                    платформенно-ориентированного метода:" + ob.i);
        ob.test(); // вызвать платформенно-ориентированный метод
        System.out.println("Содержимое переменной ob.i после вызова
                    платформенно-ориентированного метода:" + ob.i);
    }

    // объявить платформенно-ориентированный метод
    public native void test() ;

    // загрузить библиотеку DLL, содержащую статический метод
    static {
        System.loadLibrary("NativeDemo");
    }
}
```

Обратите внимание на то, что метод `test()` объявлен как `native` и не имеет тела. Это метод, который будет реализован далее на языке C. Обратите также внимание на блок оператора `static`. Как пояснялось ранее, блок, объявленный как `static`, выполняется только один раз при запуске программы, а точнее говоря, при первой загрузке ее класса. В данном случае он служит для загрузки динамически подключаемой библиотеки, которая содержит реализацию метода `test()`. (Ниже будет показано, как создать такую библиотеку.)

Динамически подключаемая библиотека загружается методом `loadLibrary()`, входящим в состав класса `System`. Общая форма этого метода приведена ниже.

```
static void loadLibrary(String имя_файла)
```

Здесь параметр *имя_файла* обозначает символьную строку, в которой задается имя файла, содержащего библиотеку. Для среды Windows предполагается, что этот файл имеет расширение **.DLL**.

Введя исходный текст рассматриваемой здесь программы, скомпилируйте ее, чтобы получить файл `NativeDemo.class`. Затем воспользуйтесь инструментальным средством **javah.exe**, чтобы создать заголовочный файл `NativeDemo.h` (это инструментальное средство входит в состав JDK). Файл `NativeDemo.h` вам предстоит включить в свою реализацию метода `test()`. Чтобы получить файл `NativeDemo.h`, выполните следующую команду:

```
javah -jni NativeDemo
```

Эта команда создает файл `NativeDemo.h`, который должен быть включен в исходный файл C, реализующий метод `test()`. Ниже приведен результат выполнения приведенной выше команды.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeDemo */

#ifndef _Included_NativeDemo
#define _Included_NativeDemo
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class: NativeDemo
 * Method: test
```

```
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Обратите особое внимание на следующую строку кода, которая определяет прототип создаваемой вами функции `test()`:

```
JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *, jobject);
```

Следует иметь в виду, что функция будет называться `Java_NativeDemo_test()`. Именно так и следует называть реализуемую вами платформенно-ориентированную функцию. Это означает, что вместо функции `test()` вы создаете на C функцию `Java_NativeDemo_test()`. Часть `NativeDemo` в префиксе добавляется, поскольку она обозначает, что метод `test()` является членом класса `NativeDemo`. Не следует забывать, что в другом классе можно объявить свой метод `test()`, совершенно не похожий на тот, который объявлен в классе `NativeDemo`. Поэтому для различения разных версий этого метода в его имя в качестве префикса включается имя класса. Как правило, платформенно-ориентированным функциям присваивается имя, содержащее в качестве префикса имя класса, в котором он объявлен.

Создав требующийся заголовочный файл, можете написать свою реализацию метода `test()` и сохранить ее в файле `NativeDemo.c`. Ниже приведен пример реализации этого метода на C.

```
/* Этот файл содержит версию метода test() на C.*/

#include <jni.h>
#include "NativeDemo.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *env, jobject obj)
{
    jclass cls;
    jfieldID fid;
    jint i;

    printf("Запуск платформенно-ориентированного метода.\n");
    cls = (*env)->GetObjectClass(env, obj);
    fid = (*env)->GetFieldID(env, cls, "i", "I");

    if(fid == 0) {
        printf("Невозможно получить поле id.\n");
        return;
    }
    i = (*env)->GetIntField(env, obj, fid);
    printf("i = %d\n", i);
    (*env)->SetIntField(env, obj, fid, 2*i);
    printf("Завершение платформенно-ориентированного метода.\n");
}
```

Как видите, этот файл включает в себя заголовок `jni.h`, содержащий интерфейсную информацию. Этот файл поставляется вместе с компилятором Java. Напомним, что заголовочный файл `NativeDemo.h` создан ранее по команде **`javah`**.

В этой функции метод `GetObjectClass()` служит для получения структуры C, содержащей данные о классе `NativeDemo`. Метод `GetFieldID()` возвращает структуру C, содержащую данные о поле `i` из этого класса. А метод `GetIntField()` извлекает исходное значение этого поля и сохраняет его обновленное значение (дополнительные методы, управляющие другими типами данных, см. в файле `jni.h`).

Создав исходный файл `NativeDemo.c`, его следует скомпилировать, а затем сформировать динамически подключаемую библиотеку. Для того чтобы сделать это с помощью компилятора Microsoft C/C++, введите в командной строке следующую команду, дополнительно указав, если потребуется, путь к файлу `jni.h` и его подчиненному файлу `jni_md.h`:

```
Cl /LD NativeDemo.c
```

По этой команде будет создан файл `NativeDemo.dll`. И только после этого можете запустить программу на Java, которая выдаст следующий результат:

```
Содержимое переменной ob.i перед вызовом
платформенно-ориентированного метода: 10
Запуск платформенно-ориентированного метода.
i = 10
Завершение платформенно-ориентированного метода.
Содержимое переменной ob.i после вызова
платформенно-ориентированного метода: 20
```

Трудности, связанные с платформенно-ориентированными методами

Платформенно-ориентированные методы выглядят многообещающе, поскольку позволяют получить доступ к существующей базе библиотечных подпрограмм, а также надеяться на высокую скорость работы программ. Но с этими методами связаны две значительные трудности.

- **Потенциальный риск нарушения безопасности.** Платформенно-ориентированный метод выполняет конкретный машинный код, поэтому он может получить доступ к любой части системы. Это означает, что платформенно-ориентированный код не ограничивается только исполняющей средой Java, а следовательно, он несет в себе, например, угрозу заражения вирусами. Именно по этой причине в аплетах нельзя использовать платформенно-ориентированные методы. Кроме того, загрузка динамически подключаемых библиотек может быть ограничена и произведена только с разрешения диспетчера защиты.
- **Потеря переносимости.** Платформенно-ориентированный код содержится в динамически подключаемой библиотеке, поэтому он должен быть представлен на той машине, которая выполняет программу на Java. Более того, каждый платформенно-ориентированный метод зависит от конкретного процессора и операционной системы, а следовательно, каждая динамически подключаемая библиотека оказывается непереносимой. Таким

образом, прикладная программа на Java, в которой применяются платформенно-ориентированные методы, может выполняться только на той машине, на которой установлена совместимая динамически подключаемая библиотека.

Применение платформенно-ориентированных методов должно быть ограничено, поскольку они делают прикладную программу на Java непереносимой и представляют существенный риск нарушения безопасности.

Применение ключевого слова `assert`

Еще одним относительно новым дополнением языка Java является ключевое слово `assert`. Оно используется на стадии разработки программ для создания так называемых *утверждений* — условий, которые должны быть истинными во время выполнения программы. Например, в программе может быть метод, который всегда возвращает положительное целое значение. Его можно проверить утверждением, что возвращаемое значение больше нуля, используя оператор `assert`. Если во время выполнения программы условие оказывается истинным, то никаких действий больше не выполняется. Но если условие окажется ложным, то генерируется исключение типа `AssertionError`. Утверждения часто применяются с целью проверить, что некоторое ожидаемое условие действительно выполняется. В коде окончательной версии программы утверждения, как правило, отсутствуют.

Ключевое слово `assert` имеет две формы. Первая его форма выглядит следующим образом:

```
assert условие;
```

где *условие* обозначает выражение, в результате вычисления которого должно быть получено логическое значение. Если это логическое значение `true`, то утверждение истинно и никаких действий больше не выполняется. Если же вычисление условия дает логическое значение `false`, то утверждение не подтверждается и по умолчанию генерируется объект исключения типа `AssertionError`.

Ниже приведена вторая форма оператора `assert`.

```
assert условие: выражение;
```

В этой версии *выражение* обозначает значение, которое передается конструктору класса исключения `AssertionError`. Это значение преобразуется в строковую форму и выводится, если утверждение не подтверждается. Как правило, в качестве *выражения* задается символьная строка, но, в общем, разрешается любое выражение, кроме типа `void`, при условии, что оно допускает приемлемое строковое преобразование.

Ниже приведен пример программы, демонстрирующий применение оператора `assert`. В этом примере проверяется, что метод `getnum()` возвращает положительное значение.

```
// Продемонстрировать применение оператора assert
class AssertDemo {
```

```
    static int val = 3;

    // возвратить целочисленное значение
    static int getnum() {
        return val--;
    }

    public static void main(String args[])
    {
        int n;

        for(int i=0; i < 10; i++) {
            n = getnum();

            assert n > 0; // не подтвердится, если n == 0

            System.out.println("n равно " + n);
        }
    }
}
```

Чтобы разрешить проверку утверждений во время выполнения, следует указать параметр **-ea** в командной строке. Например, для проверки утверждений в классе `AssertDemo` нужно ввести следующую команду:

```
java -ea AssertDemo
```

После компиляции и запуска только что описанным образом эта программа выводит следующий результат:

```
n равно 3
n равно 2
n равно 1
Exception in thread "main" java.lang.AssertionError
    at AssertDemo.main(AssertDemo.java:17)
<Исключение в потоке "main" java.lang.AssertionError
    при вызове AssertDemo.main(AssertDemo.java:17)>
```

В методе `main()` выполняются повторяющиеся вызовы метода `getnum()`, возвращающего целочисленное значение. Это значение присваивается переменной `n`, а затем проверяется в операторе `assert`, как показано ниже.

```
assert n > 0; // не подтвердится, если n == 0
```

Исход вычисления этого оператора окажется неудачным, т.е. утверждение не подтвердится, если значение переменной `n` будет равно нулю, что произойдет после четвертого вызова метода `getnum()`. И когда это произойдет, будет сгенерировано исключение.

Как пояснялось выше, имеется возможность задать сообщение, которое выводится, если утверждение не подтверждается. Так, если подставить следующее утверждение в исходный код из предыдущего примера программы:

```
assert n > 0 : "n отрицательное!";
```

то будет выдан такой результат:

```
n равно 3
n равно 2
n равно 1
```

```
Exception in thread "main" java.lang.AssertionError : n отрицательное!
    at AssertDemo.main(AssertDemo.java:17)
```

Для правильного понимания утверждений важно иметь в виду следующее: на них нельзя полагаться для выполнение каких-нибудь конкретных действий в программе. Дело в том, что отлаженный код окончательной версии программы будет выполняться с отключенным режимом проверки утверждений. Рассмотрим в качестве примера следующий вариант предыдущей программы:

```
// Неудачное применение оператора assert!!!
class AssertDemo {
    // получить генератор случайных чисел
    static int val = 3;

    // возвратить целочисленное значение
    static int getnum() {
        return val--;
    }

    public static void main(String args[])
    {
        int n = 0;

        for(int i=0; i < 10; i++) {

            assert (n = getnum()) > 0; // Неудачная идея!

            System.out.println("n is " + n);
        }
    }
}
```

В этой версии программы вызов метода `getnum()` перенесен в оператор `assert`. И хотя такой прием оказывается вполне работоспособным, когда активизирован режим проверки утверждений, отключение этого режима приведет к неправильной работе программы, потому что вызов метода `getnum()` так и не произойдет! По существу, значение переменной `n` должно быть теперь инициализировано, поскольку компилятор выявит, что это значение может и не быть присвоено в операторе `assert`.

Утверждения являются полезным нововведением в Java, потому что они упрощают проверку ошибок, часто выполняемую на стадии разработки. Так, если до появления утверждений нужно было проверить, что переменная `n` имеет в приведенном выше примере программы положительное значение, то пришлось были написать последовательность кода, аналогичную следующей:

```
if(n < 0) {
    System.out.println("n отрицательное!");
    return; // или сгенерировать исключение
}
```

А с появлением утверждений для той же самой проверки требуется только одна строка кода. Более того, строки кода с оператором `assert` не придется удалять из окончательного варианта прикладного кода.

Параметры включения и отключения режима проверки утверждений

При выполнении кода можно отключить все утверждения, указав параметр **-da**. Включить или отключить режим проверки утверждений можно для конкретного пакета (и всех его внутренних пакетов), указав его имя и три точки после параметра **-ea** или **-da**. Например, чтобы включить режим проверки утверждений в пакете `MyPack`, достаточно ввести следующее:

```
-ea:MyPack...
```

А для того чтобы отключить режим проверки утверждений, ввести следующее:

```
-da:MyPack...
```

Кроме того, класс можно указать с параметром **-ea** или **-da**. В качестве примера ниже показано, как включить режим проверки утверждений отдельно в классе `AssertDemo`.

```
-ea:AssertDemo
```

Статический импорт

В Java имеется языковое средство, расширяющее возможности ключевого слова `import` и называемое *статическим импортом*. Оператор `import`, предваряемый ключевым словом `static`, можно применять для импорта статических членов класса или интерфейса. Благодаря статическому импорту появляется возможность ссылаться на статические члены непосредственно по именам, не уточняя их именем класса. Это упрощает и сокращает синтаксис, требующийся для работы со статическими членами.

Чтобы стала понятнее польза от статического импорта, начнем с примера, в котором он *не* используется. В приведенной ниже программе вычисляется гипотенуза прямоугольного треугольника. С этой целью вызываются два статических метода из встроенного в Java класса `Math`, входящего в пакет `java.lang`. Первый из них — метод `Math.pow()` — возвращает числовое значение, возведенное в указанную степень, а второй — метод `Math.sqrt()` — возвращает квадратный корень числового значения аргумента.

```
// Вычислить длину гипотенузы прямоугольного треугольника
class Hypot {
    public static void main(String args[]) {
        double side1, side2;
        double hypot;
        side1 = 3.0;
        side2 = 4.0;

        // Обратите внимание на то, что имена методов sqrt() и pow()
        // должны быть уточнены именем их класса - Math
        hypot = Math.sqrt(Math.pow(side1, 2) +
                          Math.pow(side2, 2));

        System.out.println("При заданной длине сторон " +
```

```
                        side1 + " и " + side2 +
                        "гипотенуза равна " + hypot);
    }
}
```

Методы `pow()` и `sqrt()` являются статическими, поэтому они должны быть вызваны с указанием имени их класса — `Math`. Это приводит к следующему громоздкому коду вычисления гипотенузы:

```
hypot = Math.sqrt(Math.pow(side1, 2) +
                  Math.pow(side2, 2));
```

Как показывает данный простой пример, очень неудобно указывать каждый раз имя класса при вызове методов `pow()` и `sqrt()` или любых других встроенных в Java методов, выполняющих математические операции наподобие `sin()`, `cos()` и `tan()`.

Подобных неудобств можно избежать, если воспользоваться статическим импортом, как показано в приведенной ниже версии программы из предыдущего примера.

```
// Воспользоваться статическим импортом для доступа
// к встроенным в Java методам sqrt() и pow()
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

// вычислить гипотенузу прямоугольного треугольника
class Hypot {
    public static void main(String args[]) {
        double side1, side2;
        double hypot;

        side1 = 3.0;
        side2 = 4.0;

        // Здесь методы sqrt() и pow() можно вызывать
        // непосредственно, опуская имя их класса
        hypot = sqrt(pow(side1, 2) + pow(side2, 2));

        System.out.println("При заданной длине сторон " +
                           side1 + " и " + side2 +
                          " гипотенуза равна " + hypot);
    }
}
```

В этой версии программы имена методов `sqrt()` и `pow()` становятся видимыми благодаря оператору статического импорта, как показано ниже.

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

После этих операторов больше нет нужды уточнять имена методов `pow()` и `sqrt()` именем их класса. Таким образом, вычисление гипотенузы может быть выражено более удобным способом, как показано ниже. Как видите, эта форма не только упрощает код, но и делает его более удобочитаемым.

```
hypot = sqrt(pow(side1, 2) + pow(side2, 2));
```

Имеются две основные формы оператора `import static`. Первая форма, употреблявшаяся в предыдущем примере программы, делает видимым единственное имя. В общем виде эта форма статического импорта такова:

```
import static пакет.имя_типа.имя_статического_члена;
```

где *имя_типа* обозначает имя класса или интерфейса, который содержит требуемый статический член. Полное имя его пакета указано в части *пакет*, а имя члена — в части *имя_статического_члена*.

Вторая форма статического импорта позволяет импортировать все статические члены данного класса или интерфейса. В общем виде эта форма выглядит следующим образом:

```
import static пакет.имя_типа.*;
```

Если предполагается применять много статических методов или полей, определенных в классе, то эта форма позволяет сделать их доступными, не указывая каждый из них в отдельности. Так, в предыдущем примере программы с помощью единственного оператора `import` можно сделать доступными методы `pow()` и `sqrt()`, а также *все остальные* статические члены класса `Math`, как показано ниже.

```
import static java.lang.Math.*;
```

Разумеется, статический импорт не ограничивается только классом `Math` или его методами. Например, в следующей строке становится доступным статическое поле `System.out`.

```
import static java.lang.System.out;
```

После этого оператора данные можно выводить на консоль, не уточняя стандартный поток вывода `out` именем его класса `System`, как показано ниже.

```
out.println("Импортировав стандартный поток вывода System.out, " +
            "можно пользоваться им непосредственно.");
```

Тем не менее приведенный выше способ импорта стандартного потока вывода `System.out` столь же удобен, сколь и полемичен. Несмотря на то что такой способ сокращает исходный текст программы, тем, кто его читает, не вполне очевидно, что `out` обозначает `System.out`. Следует также иметь в виду, что, помимо импорта статических членов классов и интерфейсов, определенных в прикладном программном интерфейсе Java API, импортировать статическим способом можно также статические члены своих собственных классов и интерфейсов.

Каким бы удобным ни был статический импорт, очень важно не злоупотреблять им. Не следует забывать, что библиотечные классы Java объединяются в пакеты для того, чтобы избежать конфликтов пространств имен и непреднамеренного сокрытия прочих имен. Если статический член используется в прикладной программе только один или два раза, то его лучше *не* импортировать. К тому же некоторые статические имена, как, например, `System.out`, настолько привычны и узнаваемы, что их вряд ли стоит вообще импортировать. Статический импорт следует оставить на тот случай, если статические члены применяются многократно, как, например, при выполнении целого ряда математических вы-

числений. В сущности, этим языковым средством стоит пользоваться, но только не злоупотреблять им.

Вызов перегружаемых конструкторов по ссылке this()

Пользуясь перегружаемыми конструкторами, иногда удобно вызывать один конструктор из другого. Для этого в Java имеется еще одна форма ключевого слова this. В общем виде эта форма выглядит следующим образом:

this(*список_аргументов*)

По ссылке this() сначала выполняется перегружаемый конструктор, который соответствует заданному *списку_аргументов*, а затем — любые операторы, находящиеся в теле исходного конструктора, если таковые имеются. Вызов конструктора по ссылке this() должен быть первым оператором в конструкторе.

Чтобы стало понятнее, как пользоваться ссылкой this(), обратимся к краткому примеру. Рассмотрим сначала приведенный ниже пример класса, в котором ссылка this() *не* используется.

```
class MyClass {
    int a;
    int b;

    // инициализировать поля a и b по отдельности
    MyClass(int i, int j) {
        a = i;
        b = j;
    }

    // инициализировать поля a и b одним и тем же значением
    MyClass(int i) {
        a = i;
        b = i;
    }

    // присвоить полям a и b нулевое значение по умолчанию
    MyClass( ) {
        a = 0;
        b = 0;
    }
}
```

Этот класс содержит три конструктора, каждый из которых инициализирует значения полей a и b. Первому конструктору передаются отдельные значения для инициализации полей a и b. Второй конструктор принимает только одно значение и присваивает его обоим полям, a и b. А третий присваивает полям a и b нулевое значение по умолчанию.

Используя ссылку this(), приведенный выше класс MyClass можно переписать следующим образом:

```
class MyClass {
    int a;
    int b;

    // инициализировать поля a и b по отдельности
    MyClass(int i, int j) {
        a = i;
        b = j;
    }

    // инициализировать поля a и b одним и тем же значением
    MyClass(int i) {
        this(i, i); // по этой ссылке вызывается
                    // конструктор MyClass(i, i);
    }

    // присвоить полям a и b нулевое значение по умолчанию
    MyClass( ) {
        this(0); // а по этой ссылке вызывается
                 // конструктор MyClass(0)
    }
}
```

В этой версии класса `MyClass` значения полям `a` и `b` в действительности присваиваются только в конструкторе `MyClass(int, int)`. А два других конструктора просто вызывают первый конструктор (прямо или косвенно) по ссылке `this()`. Рассмотрим, например, что произойдет, если выполнить следующий оператор:

```
MyClass mc = new MyClass(8);
```

Вызов конструктора `MyClass(8)` приводит к выполнению ссылки `this(8,8)`, которая преобразуется в вызов конструктора `MyClass(8,8)`, поскольку именно эта версия конструктора класса `MyClass` соответствует списку аргументов, передаваемому по ссылке `this()`. А теперь рассмотрим следующий оператор, в котором используется конструктор по умолчанию:

```
MyClass mc2 = new MyClass();
```

В данном случае происходит обращение по ссылке `this(0)`, приводящее к вызову конструктора `MyClass(0)`, поскольку именно эта версия конструктора соответствует списку параметров. Разумеется, конструктор `MyClass(0)` вызывает далее конструктор `MyClass(0,0)`, как пояснялось выше.

Одной из причин, по которой стоит вызывать перегружаемые конструкторы по ссылке `this()`, служит потребность избежать дублирования кода. Зачастую сокращение дублированного кода ускоряет загрузку классов, поскольку объектный код становится компактнее. Это особенно важно для программ, доставляемых через Интернет, когда время их загрузки критично. Применение ссылки `this()` позволяет также оптимально структурировать прикладной код, когда конструкторы содержат большой объем дублированного кода.

Следует, однако, иметь в виду, что конструкторы, вызывающие другие конструкторы по ссылке `this()`, выполняются медленнее, чем те, что содержат весь код, необходимый для инициализации. Дело в том, что механизм вызова и возвращения, используемый при вызове второго конструктора, требует дополнительных издержек. Если класс предполагается употреблять для создания

небольшого количества объектов или если его конструкторы, вызывающие друг друга по ссылке `this()`, будут использоваться редко, то снижение производительности во время выполнения, скорее всего, окажется незначительным. Но если во время выполнения программы предполагается создание большого количества (порядка тысяч) объектов такого класса, то дополнительные издержки, связанные с вызовом одних конструкторов из других по ссылке `this()`, могут отрицательно сказаться на производительности. Создание объектов затрагивает всех пользователей такого класса, и поэтому придется тщательно взвесить преимущества более быстрой загрузки в сравнении с увеличением времени на создание объекта.

Следует также иметь в виду, что вызов очень коротких конструкторов, как, например, из класса `MyClass`, по ссылке `this()` зачастую лишь незначительно увеличивает размер объектного кода. (В некоторых случаях никакого уменьшения объема объектного кода вообще не происходит.) Дело в том, что байт-код, который устанавливается и возвращается из вызова конструктора по ссылке `this()`, добавляет инструкции в объектный файл. Поэтому в таких случаях вызов конструктора по ссылке `this()`, несмотря на исключение дублирования кода, не даст значительной экономии времени загрузки, но может повлечь за собой дополнительные издержки на создание каждого объекта. Поэтому применение ссылки `this()` больше всего подходит для вызова тех конструкторов, которые содержат большой объем кода инициализации, а не тех, которые просто устанавливают значения в нескольких полях.

Вызывая конструкторы по ссылке `this()`, следует учитывать следующее. Во-первых, при вызове конструктора по ссылке `this()` нельзя использовать переменные экземпляра класса этого конструктора. И во-вторых, в одном и том же конструкторе нельзя использовать ссылки `super()` и `this()`, поскольку каждая из них должна быть первым оператором в конструкторе.

Компактные профили Java API

В версии JDK 8 внедрено средство, позволяющее организовать подмножества библиотеки прикладных программных интерфейсов API в так называемые *компактные профили*. Они обозначаются следующим образом: `compact1`, `compact2` и `compact3`. Каждый такой профиль содержит подмножество библиотеки. Более того, компактный профиль `compact2` включает в себя весь профиль `compact1`, а компактный профиль `compact3` — весь профиль `compact2`. Следовательно, каждый последующий компактный профиль строится на основании предыдущего. Преимущество компактных профилей заключается в том, что прикладной программе не нужно загружать библиотеку полностью. Применение компактных профилей позволяет сократить размер библиотеки, а следовательно, выполнять некоторые категории прикладных программ на тех устройствах, где отсутствует полная поддержка прикладного программного интерфейса Java API. Благодаря компактным профилям удается также сократить время, требующееся для загрузки программы. В документации на прикладной программный интерфейс Java API ука-

зывается, к какому именно элементу этого прикладного интерфейса принадлежит компактный профиль, если это вообще имеет место.

На стадии компиляции программы с помощью параметра командной строки **-profile** можно указать, какие именно элементы прикладного программного интерфейса Java API, определяемые компактным профилем, следует использовать в программе. Ниже приведена общая форма команды, по которой программу можно скомпилировать именно таким способом.

```
javac -profile имя_профиля имя_программы
```

Здесь *имя_профиля* обозначает конкретный профиль: `compact1`, `compact2` или `compact3`. Например:

```
javac -profile compact2 Test.java
```

В этом примере команды компиляции указан профиль `compact2`. Если же исходный файл программы `Test.java` содержит элементы прикладного программного интерфейса Java API, отсутствующие в компактном профиле `compact2`, то во время компиляции возникнет ошибка.