1. Ввод/вывод и апплеты

Потоки

Java-программы выполняют ввод/вывод через потоки. *Поток* производит, или потребляет информацию. Поток связывается с физическим устройством с помощью системы ввода/вывода Java (Java I/O system). Все потоки ведут себя одинаковым образом, хотя фактические физические устройства, с которыми они связаны, могут сильно различаться. Таким образом, одни и те же классы и методы ввода/вывода можно применять к устройствам любого типа. Это означает, что *поток ввода* может извлекать много различных видов входных данных: из дискового файла, с клавиатуры или сетевого разъема. Аналогично, *поток вывода* может обратиться к консоли, дисковому файлу или сетевому соединению (сокету). Благодаря потокам программа выполняет ввод/вывод, не понимая различий между клавиатурой и сетью, например. Java реализует потоки с помощью иерархии классов, определенных в пакете `java.io`.

Подход Java к потокам почти такой же, как в C/C++.

Байтовые и символьные потоки

Java 2 определяет два типа потоков: байтовый и символьный. *Байтовые потоки* предоставляют удобные средства для обработки ввода и вывода байтов. Байтовые потоки используются, например, при чтении или записи данных в двоичном коде. *Символьные потоки* предоставляют удобные средства для обработки ввода и вывода символов. Они используют Unicode и поэтому могут быть интернационализированы. Кроме того, в некоторых случаях символьные потоки более эффективны, чем байтовые.

На самом низком уровне весь ввод/вывод является байтовым. Символьно-ориентированные потоки обеспечивают удобные и эффективные средства для обработки символов.

Классы байтовых потоков

Байтовые потоки определяются в двух иерархиях классов. Наверху этой иерархии — два абстрактных класса: `InputStream` и `OutputStream`. Каждый из этих абстрактных классов имеет несколько конкретных подклассов, которые обрабатывают различия между разными устройствами, такими как дисковые файлы, сетевые соединения и даже буферы памяти. Классы байтовых потоков показаны в табл. 12.1.

Некоторые из этих классов обсуждаются далее в этом разделе. Чтобы использовать поточные классы, нужно импортировать пакет `java.io`.

Абстрактные классы `InputStream` и `OutputStream` определяют несколько ключевых методов, которые реализуются другими поточными классами. Два наиболее важных — `read()` и `write()`, которые, соответственно, читают и записывают байты данных. Оба метода объявлены как абстрактные внутри классов `InputStream` и `OutputStream` и переопределяются производными поточными классами.

Классы символьных потоков

Символьные потоки определены в двух иерархиях классов. Наверху этой иерархии два абстрактных класса: `Reader` и `Writer`. Они обрабатывают потоки символов Unicode. В Java существуют несколько конкретных подклассов каждого из них. Классы символьных потоков показаны в табл. 12.2.

Абстрактные классы `Reader` и `Writer` определяют несколько ключевых методов, которые реализуются другими поточными классами. Два самых важных метода — `read()` и `write()`, которые читают и записывают символы данных, соответственно. Они переопределяются производными поточными классами.

Таблица 12.1. Классы байтовых потоков

Поточный класс	Значение
`BufferedInputStream`	Буферизированный поток ввода
`BufferedOutputStream`	Буферизированный поток вывода
`ByteArrayInputStream`	Поток ввода, который читает из байт-массива
`ByteArrayOutputStream`	Поток вывода, который записывает в байт-массив
`DataInputStream`	Поток ввода, который содержит методы для чтения данных стандартных типов Java
`DataOutputStream`	Поток вывода, который содержит методы для записи данных стандартных типов Java
`FileInputStream`	Поток ввода, который читает из файла
`FileOutputStream`	Поток вывода, который записывает в файл
`FilterInputStream`	Реализует InputStream
`FilterOutputStream`	Реализует OutputStream
`InputStream`	Абстрактный класс, который описывает поточный ввод

OutputStream	Абстрактный класс, который описывает поточный вывод
PipedlnputStream	Канал ввода
PipedOutputStream	Канал вывода
PrintStream	Поток вывода, который поддерживает print () и println()
PushbacklnputStream	Поток (ввода), который поддерживает однобайтовую операцию "unget", возвращающую байт в поток ввода
RandomAccessFile	Поддерживает ввод/вывод файла произвольного доступа
SequencelnputStream	Поток ввода, который является комбинацией двух или нескольких потоков ввода, которые будут читаться последовательно, один за другим

Таблица 12. 2. Классы ввода/вывода символьных потоков

Поточный класс	**Значение**
BufferedReader	Буферизированный символьный поток ввода
BufferedWriter	Буферизированный символьный поток вывода
CharArrayReader	Поток ввода, который читает из символьного массива
CharArrayWriter	Выходной поток, который записывает в символьный массив
FileReader	Поток ввода, который читает из файла
FileWriter	Выходной поток, который записывает в файл
FilterReader	Отфильтрованный поток ввода
FilterWriter	Отфильтрованный поток вывода
InputStreamReader	Поток ввода, который переводит байты в символы
LineNumberReader	Поток ввода, который считает строки
OutputStreamWriter	Поток вывода, который переводит символы в байты
PipedReader	Канал ввода
PipedWriter	Канал вывода
PrintWriter	Поток вывода, который поддерживает print() и println()
PushbackReader	Поток ввода, возвращающий символы в поток ввода

`Reader`	Абстрактный класс, который описывает символьный поток ввода
`StringReader`	Поток ввода, который читает из строки
`StringWriter`	Поток вывода, который записывает в строку
`Writer`	Абстрактный класс, который описывает символьный поток вывода

1.1. Предопределенные потоки

Как известно, все программы Java автоматически импортируют пакет `java.lang`. Этот пакет определяет класс с именем `System`, инкапсулирующий некоторые аспекты исполнительной среды Java. Например, используя некоторые из его методов, можно получить текущее время и параметры настройки различных свойств, связанных с системой. Класс `System` содержит также три предопределенные поточные переменные `in`, `out` и `err`. Эти поля объявлены в `System` со спецификаторами `public` и `static`. Это означает, что они могут использоваться любой частью программы, и причем без ссылки на конкретный `System`-объект.

Объект `System.out` называют *потоком стандартного вывода.* По умолчанию с ним связана консоль. На объект `System.in` ссылаются как на *стандартный ввод,* который по умолчанию связан с клавиатурой. К объекту `System.err` обращаются как к *стандартному потоку ошибок,* который по умолчанию также связан с консолью. Однако эти потоки могут быть переназначены на любое совместимое устройство ввода/вывода.

`System.in` – это объект типа `InputStream`; `System.out` и `System.err` — объекты типа `PrintStream`. Это байтовые потоки, хотя они обычно используются, чтобы читать и записывать символы с консоли и на консоль. Однако их можно упаковать в символьные потоки, если нужно.

В предыдущих примерах использовался объект `System.out`. Аналогичным образом можно использовать и `System.err`.

Чтение консольного ввода

Java не имеет обобщенного метода консольного ввода, который соответствует стандартной С-функции `scanf()` или операциям ввода С++.

Консольный ввод в Java выполняется с помощью считывания из объекта `System.in`. Чтобы получить символьный поток, который присоединен к консоли, нужно перенести ("упаковывать") `System.in` в объект типа `BufferedReader`. Класс `BufferedReader` поддерживает

буферизированный входной поток. Обычно используется следующий его конструктор:

```
BufferedReader (Reader inputReader)
```

где inputReader — поток, который связан с создающимся экземпляром класса BufferedReader. Reader — абстрактный класс. Один из его конкретных подклассов — это InputStreamReader, который преобразует байты в символы. Чтобы получить InputStreamReader-объект, который связан с System.in, используйте следующий конструктор:

```
InputStreamReader (InputStream inputStream)
```

Поскольку System.in ссылается на объект типа InputStream, его можно использовать в качестве параметра InputStream. Следующая строка кода создает объект класса BufferedReader, который связан с клавиатурой:

```
BufferedReader br = new BufferedReader(new InputStreamReader (Sys-
tem.in));
```

После того как этот оператор выполнится, объектная переменная br станет символьным потоком, связанным с консолью через System.in.

Чтение символов

Для чтения символа из BufferedReader используйте метод read(). Версия read(), которую мы будем применять, такова:

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

При каждом вызове read() читает символ из входного потока и возвращает его в виде целочисленного значения. Когда read() сталкивается с концом потока, то возвращает -1. Он может выбрасывать исключение ввода/вывода (I/O-исключение — IOException).

Следующая программа демонстрирует read(), читая символы с консоли, пока пользователь не напечатает "q":

Программа 71. Чтение символов с консоли

```
// Файл BRRead.java
// Использует 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 завершения.");
   // Чтение символов
```

```
      do {
         c = (char) br.read();
         System.out.println(c);
      }
      while(c != 'q');
   }
}
```

Результат выполнения этого примера:

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

Чтение набранных на клавиатуре символов начинается только после нажатия клавиши <Enter> так как `System.in` по умолчанию — буферизированный поток и вводимые символы сначала помещаются в буфер, а затем уже извлекаются из буфера.

Чтение строк

Метод `readLine()` класса `BufferedReader`:

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

возвращает `String`-объект, который содержит строку, набранную на клавиатуре.

Следующая программа демонстрирует `BufferedReader` и метод `readLine()`. Она читает и отображает строки текста, пока не будет введено слово "`stop`":

Программа 72. Чтение строк с консоли

```
// Файл BRReadLines.java
// Читает строки с консоли, используя BufferedReader.
import java.io.*;
class BRReadLines {
   public static void main(String args[]) throws IOException
   {
   // Создать BufferedReader, используя System.in
      BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));
      String str;
      System.out.println("Введите строки текста.");
      System.out.println("Введите 'stop' для завершения.");
      do {
         str = br.readLine();
```

```
            System.out.println(str);
        }
        while(!str.equals("stop"));
    }
}
```

Следующий пример демонстрирует крошечный текстовый редактор. Сначала он создает массив string-объектов и затем считывает строки текста, сохраняя каждую из них в массиве. Он будет читать до сотой строки или до тех пор, пока не будет введена строка "stop". Для чтения с консоли используется объект класса BufferedReader (переменная pr).

Программа 73. Квазиредактор

```
// Файл TinyEdit.java
// Крошечный редактор.
import java.io.*;
class TinyEdit {
    public static void main(String args[]) throws IOException
    {
    // Создать BufferedReader-объект, используя System.in
        BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));
        String str[] = new String[100]; // Массив из 100 строк
        System.out.println("Введите строки текста.");
        System.out.println("Введите 'stop' для завершения.");
        for(int i = 0; i < 100; i++) {
            str[i] = br.readLine();
            if(str[i].equals("stop"))
                break;
        }
        System.out.println("\nВот ваш файл:");
    // Вывести строки на экран.
        for(int i = 0; i < 100; i++) {
            if (str[i].equals("stop"))
                break;
            System.out.println(str [i]);
        }
    }
}
```

Пример вывода этой программы:

```
Введите строки текста.
Введите 'stop' для завершения.
Это строка 1.
Это строка 2.
Java облегчает работу со строками.
Создать String-объекты.
stop
Вот ваш файл:
Это строка 1.
Это строка 2.
Java облегчает работу со строками.
```

создать String-объекты.

Запись консольного вывода

Консольный вывод легче всего выполнить с помощью описанных ранее методов print() и println(), которые используются в большинстве примеров. Эти методы определены классом PrintStream (который является типом (классом) объекта System.out). Хотя System.out – *байтовый* поток, его использование для вывода в простых программах все еще допустимо. Его *символьная* альтернатива описана в следующем разделе.

Поскольку PrintStream – выходной поток, производный от OutputStream, он также реализует метод нижнего уровня write(). Его можно использовать для записи на консоль. Самая простая форма write(), определенная в PrintStream, имеет вид:

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

Этот метод записывает в файл байт, указанный в параметре byteval. Хотя byteval объявлен как целое число, записываются только младшие восемь битов. Ниже показан короткий пример, который использует write() для вывода на экран символа "A", за которым следует символ newline:

Программа 74. Использование write()

```
// Файл WriteDemo.java
// Демонстрирует System.out.write().
class WriteDemo {
   public static void main(String args[]) {
      int b;
      b = 'A';
      System.out.write(b);
      System.out.write('\n');
      System.out.write(65);
      System.out.write('\n');
   }
}
```

Программа выводит:

```
A
A
```

Метод write() менее удобен чем print() и println().

Класс PrintWriter

Хотя использование объекта System.out для записи на консоль все еще допустимо в Java, его применение рекомендуется главным образом для отладочных целей или для демонстрационных программ. Для

реальных Java-программ для записи на консоль рекомендуется работать с потоком типа `PrintWriter`. `PrintWriter` — это один из классов символьного ввода/вывода. Использование подобного класса для консольного вывода облегчает интернационализацию программ.

`PrintWriter` определяет несколько конструкторов. Мы будем использовать следующий:

```
PrintWriter (OutputStream outputStrm, boolean flushOnNewline)
```

Здесь `outputStrm` — объект типа `OutputStream`; `flushOnNewline` — булевский параметр, используемый как средство управления сбрасыванием выходного потока в буфер вывода (на диск) каждый раз, когда выводится символ `newline` (`\n`). Если `fiushOnNewiine` — `true`, поток сбрасывается автоматически, если — `false`, то не автоматически.

`PrintWriter` поддерживает методы `print()` и `println()` для всех типов, включая `object`. Поэтому эти методы можно применять так же, как они использовались с объектом `System.out`. Если аргумент не является простым типом, то методы класса `PrintWriter` вызывают объектный метод `toString()` и затем печатают результат.

Чтобы записывать на консоль, используя класс `printWriter`, нужно создать объект `System.out` для выходного потока, и сбрасывать поток после каждого символа `newline`. Например, следующая строка кода создает объект типа `PrintWriter`, который соединен с консольным выводом:

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

Очередное приложение иллюстрирует использование `PrintWriter` для управления консольным выводом:

Программа 75. Использование printWriter

```
// Файл PrintWriterDemo.java
// Демонстрирует Printwriter.
import java.io.*;
public class PrintWriterDemo {
   public static void main(String args[])   {
      PrintWriter pw = new PrintWriter(System.out, true);
      pw.println("Это строка:");
      int i = -7;
      pw.println(i);
      double d = 4.5e-7;
      pw.println(d);
   }
}
```

Вывод этой программы:

```
Это строка:
-7
```

```
4.5E-7
```

Для отладки программы удобно использовать `System.out` для записи простого текстового вывода на консоль. Однако с использованием `PrintWriter` реальные приложения будет проще интернационализировать. Поскольку никакого преимущества от использования `PrintWriter` в демонстрационных программах не проявляется, для записи на консоль далее продолжится использование объекта `System.out`.

1.2. Чтение и запись файлов

Java обеспечивает ряд классов и методов, которые позволяют читать и записывать файлы. Для Java все файлы имеют байтовую структуру, а Java обеспечивает методы для чтения и записи байтов в файл. Кроме того, Java позволяет упаковывать *байтовый* файловый поток в символьно-ориентированный объект. Эта методика описана ниже. Здсь рассматриваются основы файлового ввода/вывода.

Для создания байтовых потоков, связанных с файлами, чаше всего используются два поточных класса — `FileInputStream` и `FileOutputStream`. Для открытия файла создается объект одного из этих классов с указанием имени файла как аргумента конструктора. Хотя оба класса поддерживают несколько переопределенных конструкторов, мы будем использовать только следующее формы:

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

где `fileName` определяет имя открываемого файла. Когда создается входной поток при отсутствующем файле, выбрасывается исключение `FileNotFoundException`. Для выходных потоков, если файл не может быть создан, выбрасывается такое же исключение (`FileNotFoundException`).

Когда выходной файл открывается, любой файл, существовавший ранее с тем же самым именем, разрушается.

После завершения работы с файлом, его нужно закрыть, вызвав метод `close()`, который определен как в `FileInputStream`, так и в `FileOutputStream` в следующей форме:

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

Для чтения файла можно использовать версию метода `read()`, который определен в `FileInputStream`. Мы будем использовать такую версию:

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

При каждом вызове метод читает один байт из файла и возвращает его в форме целочисленного значения. Когда read() встречает символ конца файла (EOF), то возвращает -1. Метод read() может выбрасывать исключение IOException.

Следующая программа использует read() для ввода и отображения содержимого текстового файла, имя которого указывается как параметр командной строки. Обратите внимание на блоки try/catch, обрабатывающие две ошибки, которые могут произойти во время использования программы: указанный файл не найден, или пользователь забыл включить в командную строку имя файла.

Программа 76. Чтение файла

```
// Файл ShowFile.java
/* Выведет на экран текстовый файл.
При запуске программы укажите  (в параметре команды запуска) имя файла,
который вы хотите просмотреть. Например, чтобы просмотреть файл с именем
TEST.TXT, Используйте следующую командную строку
Java ShowFile TEST.TXT
*/
import java.io.*;
class ShowFile {
   public static void main(String args[]) throws IOException
   {
      int i;
      FileInputStream fin;
      try {
         fin = new FileInputStream(args[0]);
      }
      catch(FileNotFoundException e) {
         System.out.println("Файл не найден");
         return;
      }
      catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Используйте: ShowFile имя_файла");
         return;
      }
   // Читать символы файла, пока не встретится символ EOF
      do {
         i = fin.read();
         if(i != -1)
            System.out.print((char) i);
      }
      while(i != -1);
      fin.close();
   }
}
```

Для записи в файл используется метод write(), определенный в классе FileOutputStream. Его самая простая форма имеет вид:

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

Данный метод записывает в файл байт, указанный в параметре byteval. Хотя byteval объявлен как целое число, в файл записывается только восемь младших битов. Если во время записи происходит ошибка, выбрасывается исключение IOException. В следующем примере метод write() применяется для копирования текстового файла:

Программа 77. Копирование файлов

```
// ФАйл CopyFile.java
/*
Копирование текстового файла.
При запуске этой программы укажите имя исходного файла
и целевого файла(в который будет выполняться копирование).
Например, чтобы скопировать файл с именем FIRST.TXT
в файл с именем SECOND.TXT, используйте следующую командную строку
java CopyFile FIRST.TXT SECOND.TXT
*/
import java.io.*;
class CopyFile {
   public static void main(String args[]) throws IOException
   {
      int i;
      FileInputStream fin;
      FileOutputStream fout;
      try {
   // Открыть файл для ввода
         try {
            fin = new FileInputStream(args[0]);
         }
         catch(FileNotFoundException e) {
            System.out.println("Исходный файл “ + args[0] + “ не найден");
            return;
         }
   // Открыть файл для вывода
         try {
            fout = new FileOutputStream(args[1]);
         }
         catch(FileNotFoundException e) {
            System.out.println("Ошибка открытия выходного файла " +
                                 args[1]);
            return;
         }
      }
      catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("CopyFile копирует исходный файл в выходной");
         return;
      }
   // Копировать файл
      try {
         do {
            i = fin.read();
            if(i != -1)
               fout.write(i);
```

```
         }
         while(i != -1);
      }
      catch(IOException e) {
         System.out.println("Файловая ошибка");
      }
      fin.close();
      fout.close();
   }
}
```

Обратите внимание на способ обработки потенциальных ошибок ввода/вывода в этой и в предыдущей программе. В отличие от большинства других машинных языков, включая С и C++, которые используют коды ошибки, чтобы сообщать о файловых ошибках, Java используют собственный механизм обработки особых ситуаций (исключений). Это не только делает обработку файла более ясной, но и позволяет во время ввода легко отличить состояние конца файла от файловых ошибок. В C/C++ многие функции ввода возвращают одно и то же значение, когда происходит ошибка и когда достигнут конец файла (т. е. в C/C++ признак конца файла (`EOF`) часто отображается в то же значение, что и ошибка ввода). Это обычно означает, что программист должен включить в код дополнительные операторы для того, чтобы определить, что же фактически произошло. В Java ошибки передаются в программу через исключения, а не через значения, возвращаемые методом `read()`. Таким образом, когда `read()` возвращает — 1, это означает только одно — при чтении встретился конец файла.

1.3. Апплеты. Основы программирования

Все предшествующие примеры были Java-приложениями. Однако *приложение* — это только один тип Java-программ. Другой тип программ представлен *апплетом.* Апплеты - это небольшие приложения, которые доступны на Internet-сервере, транспортируются по Internet, автоматически устанавливаются и выполняются как часть Web-документа. После того, как апплет прибывает к клиенту, он имеет ограниченный доступ к ресурсам системы, которые использует для создания произвольного мультимедийного интерфейса пользователя и выполнения комплексных вычислений без риска заражения вирусами или нарушения целостности данных.

Многие проблемы, связанные с созданием и использованием апплетов, обсуждаются ниже при рассматрении пакета `applet`**.** Здесь же представлены основные принципы, связанные с созданием апплетов, потому что они не структурированы таким же образом, как используемые до настоящего времени программы. Апплеты отличаются от приложений в нескольких ключевых областях.

Начнем с примера простого апплета:

Программа 78. Простейший апплет

```
// Файл SimpleApplet.java
import java.awt.*;
import java.applet.*;
   public class SimpleApplet extends Applet {
      public void paint (Graphics g) {
         g.drawString("A Simple Applet", 20, 20); // Вывод строки в окно
}
```

Апплет начинается с двух операторов `import`. Первый импортирует AWT классы (из большой иерархии awt-пакетов Java) (AWT - Abstract Windowing Toolkit - абстрактный оконный интерфейс.). Таким образом, апплеты взаимодействуют с пользователем через AWT, а не через классы консольного ввода/вывода. AWT осуществляет поддержку графического оконного интерфейса. Система AWT классов достаточно велика и сложна. В приведенном простом апплете использование AWT очень ограничено. Второй оператор `import` импортирует пакет `java.applet`, который содержит класс `Applet`. Каждый создаваемый апплет должен быть подклассом этого класса.

Следующая строка в программе объявляет класс `SimpleApplet`. Он должен быть объявлен как `public`, потому что к нему необходимо обеспечить доступ из кодов, которые находятся вне программы.

Внутри `SimpleApplet` объявлен метод `paint()`. Этот метод определен в AWT и должен быть переопределен апплетом. Метод `paint()` вызывается каждый раз, когда апплет должен восстанавливать изображение своего вывода. Данная ситуация может возникать в нескольких случаях. Например, окно, в котором выполняется апплет, может быть перекрыто другим окном, которое затем закрывается. Или окно апплета может быть свернуто и затем восстановлено. Метод `paint()` вызывается также, когда апплет начинает выполнение. Безотносительно причины, всякий раз, когда апплет должен перерисовать свой вывод, вызывается метод `paint()`. Метод имеет один параметр, типа `Graphics`, через который получает графический контекст, описывающий графическую среду выполнения апплета. Этот контекст используется всякий раз, когда апплету требуется вывод.

Внутри `paint()` находится обращение к методу `drawsString()`, который является членом класса `Graphics`. Этот метод выводит строку, начиная с указанных его аргументами `(x, y)`-координат в окне апплета. Он имеет следующую общую форму:

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

Здесь message — строка, которую нужно вывести. В окне апплета левый верхний угол имеет координаты (0, 0). Обращение к drawString() в апплете отображает сообщение "A Simple Applet", начиная с координат (20, 20). Координаты окна апплета измеряются в пикселах.

Заметим, что апплет не содержит метода main(). В отличие от Java-программ, апплеты не начинают выполнение в main(). Фактически, большинство апплетов даже не имеют этого метода. Апплет начинает выполнение, когда имя его класса передается программе просмотра апплетов или браузеру.

После ввода исходного кода SimpleApplet откомпилируйте его так же, как вы компилировали программы. Существует два способа выполнения апплета:

Выполнение апплета Java-совместимом Web-браузером, типа Netscape Navigator или Microsoft Internet Explorer.

Использование программы просмотра апплетов, типа стандартной утилиты JDK appletviewer. Программа просмотра апплетов выполняет апплет в его окне. Это, вообще, самый быстрый и простой способ проверки работы апплета.

Для выполнения апплета в Web-браузере нужно записать короткий текстовый файл в формате языка HTML (HyperText Markup Language) — язык разметки гипертекста), который содержит специальный тег <applet>. Следующий HTML-файл выполняет SimpleApplet:

```
<html>
   <body>
    <applet code = SimpleApplet.class width = "500" height = "100" >
    </applet>
  </body>
</html>
```

Внутри тега <applet> его параметры width и height определяют размеры окна апплета. После создания файла, можно запустить браузер и затем загрузить этот файл, что приведет к выполнению SimpleApplet.

Для выполнения SimpleApplet с помощью программы просмотра апплетов можно использовать этот же HTML-файл. Например, если предшествующий HTML-файл назвать RunApp.html, то следующая командная строка выполнит SimpleApplet:

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

Это команда создает окно, показанное на рис. 1

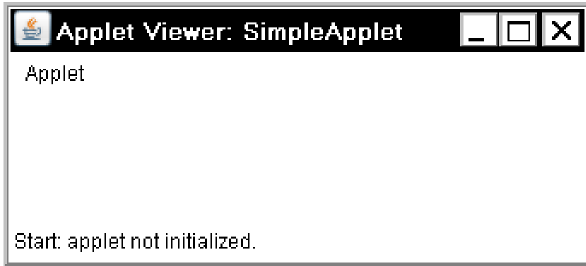

Рис. 1.Окно апплета

Однако существует и более удобный метод, с помощью которого можно ускорить тестирование. Для этого нужно просто включить в заголовок файла исходного кода Java комментарий, который содержит HTML-тег `<applet>`. Это документирует исходный код прототипом необходимых инструкций HTML, и вы можете тестировать откомпилированный апплет, просто запуская программу просмотра апплетов с файлом исходного кода Java (в качестве операнда). При использовании этого метода исходный файл `SimpleApplet.java` выглядит так:

Программа 79. Использование комментария для запуска апплета

```
// Файл SimpleApplet1.java

import java.awt.*;
import java.applet.*;
/*
<applet code = "SimpleApplet" width = 200 height = 60>
</applet>
*/
public class SimpleApplet extends Applet {
   public void paint(Graphics g) {
   g.drawString("A Simple Applet", 20, 20);

   }

}
```

Процедура быстрой разработки апплета по этой методике включает три шага.

1. Редактирование исходного файла Java.
2. Компиляция программы.
3. Запуск программы просмотра апплетов со спецификацией имени исходного файла апплета в ее аргументе. Встретив в комментарии тег `<applet>`, утилита просмотра выполнит его.

Несколько ключевых моментов нужно запомнить.

□ Апплеты не нуждаются в методе `main()`.

□ Апплеты должны выполняться программой просмотра апплетов или браузером, поддерживающим Java.

□ Пользовательский ввод/вывод в апплетах не выполняется с помощью Java-классов поточного ввода/вывода. Вместо этого апплеты используют интерфейс, обеспеченный системой AWT.

1.4. Модификаторы transient и volatile

В Java определено два интересных модификатора типа: `transient` и `volatile`. Эти модификаторы используются для обработки некоторых специальных ситуаций.

Когда экземплярная переменная объявлена как `transient`, то ее значение не будет запомнено при сохранении объекта. Например:

```
class T {
   transient int a;      // Не будет сохраняться
   int b;                // Будет сохраняться
}
```

Если бы объект типа `T` записывался в постоянную область памяти, содержимое переменной `a` не было бы сохранено, в то время как содержимое `b` — было бы.

Модификатор `volatile` сообщает компилятору, что переменная, модифицированная с его помощью, может быть неожиданно изменена другими частями программы. Одна из этих ситуаций включает многопоточные программы. В многопоточной программе два или несколько потоков иногда совместно используют одну и ту же экземплярную переменную. По соображениям эффективности, каждый поток может хранить свою собственную, частную копию такой разделяемой переменной. Реальная (или главная — `master`-) копия переменной модифицируется в разные моменты времени, например, при входе в синхронизированный метод. Такой подход обычно работает прекрасно, но время от времени может быть неэффективным. В некоторых случаях, все, что действительно имеет значение, так это то, что `master`-копия переменной всегда отражает ее текущее состояние. Для гарантии подобной ситуации просто специфицируют переменную как `volatile`, что сообщает компилятору, что он должен всегда использовать `master`-копию `volatile`-переменной (или, по крайней мере, всегда сохранять любые частные копии, соответствующие текущим значениям главной копии, и наоборот).

Модификатор `volatile` в `Java` имеет примерно то же значение, что и в `C/C++`.

1.5. Использование instanceof

Иногда полезно распознавать тип объекта во время выполнения. Например, можно иметь один поток выполнения для генерации различных типов объектов, а другой — для их обработки. В этой ситуации обрабатывающему процессу полезно было бы знать тип каждого объекта, принимаемого на обработку. Другая ситуация, в которой знание типа объекта во время выполнения очень важно, — это приведение типов (cast). В Java недопустимое приведение вызывает ошибку времени выполнения. Много недопустимых приведений можно перехватить во время компиляции. Однако операции приведения, связанные с типами объектов (т. е. с иерархиями классов), могут оказаться недопустимыми и могут быть обнаружены только во время выполнения. Например, суперкласс с именем A может иметь два подкласса: B и C. Приведение объектов B или C к типу A — законно, а приведение B-объекта к типу C (или, наоборот, C-объекта к типу B) — нет. Поскольку объект типа A может ссылаться как на объект B, так и на объект C, то каким образом вы можете определить (во время выполнения), на объект какого типа фактически ссылается A-объект перед попыткой приведения его типа к C? Это может быть объект любого типа — A, B или C. Если это — объект типа B, будет выброшено исключение (времени выполнения). Для ответа на этот вопрос Java использует специальную операцию — instanceof. Операция instanceof имеет следующую общую форму:

```
object instanceof type
```

где object — экземпляр класса; type — класс (как тип). Если object-операнд имеет тип или его тип может быть приведен к типу, указанный в type-операнде, то результат операции instanceof имеет значение true. Иначе, ее результат — false. Таким образом, instanceof — это средство, с помощью которого программа может получить информацию о типе объекта во время выполнения. Следующая программа демонстрирует операцию instanceof:

Программа 80. Проверка соответствия типов

```
// ФАйл InstanceOf.java
// Демонстрирует операцию 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 это экземпляр класса A");
      if(b instanceof B)
         System.out.println("b это экземпляр класса B");
      if(c instanceof C)
         System.out.println("c это экземпляр класса'C");
      if(c instanceof A)
         System.out.println ("тип c можно привести к типу A");
      if(a instanceof C)
         System.out.println ("тип a можно привести к типу C");
      System.out.println();
   // Сравнить производные типы
      A ob;
      ob = d;      // Ссылка на d
      System.out.println("ob теперь ссылается на d");
      if(ob instanceof D)
         System.out.println("ob теперь экземпляр класса D");
      System.out.println();
      ob = c;      // Ссылка на c
      System.out.println("ob теперь ссылается на c");
      if(ob instanceof D)
         System.out.println ("тип ob можно привести к типу D");
      else
         System.out.println("тип ob нельзя привести к типу D");
      if(ob instanceof A)
         System.out.println ("тип ob можно привести к типу A");
      System.out.println();
   // Все объекты можно привести к типу Object
      if(a instanceof Object)
         System.out.println ("тип a можно привести к типу Object");
      if(b instanceof Object)
         System.out.println ("тип b'можно привести к типу Object");
      if(c instanceof Object)
         System.out.println ("тип c можно привести к типу Object");
      if(d instanceof Object)
         System.out.println ("тип d можно привести к типу Object");
   }
}
```

Вывод этой программы:

```
a это экземпляр класса A
b это экземпляр класса B
c это экземпляр класса'C
тип c можно привести к типу A
```

```
ob теперь ссылается на d
ob теперь экземпляр класса D

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

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

Операция `instanceof` не нужна для большинства программ, потому что, вообще-то, вы знаете тип объекта, с которым работаете. Однако она может быть очень полезна, когда вы пишете обобщенные подпрограммы, работающие на объектах из сложной иерархии классов.

1.6. Ключевое слово strictfp

Java 2 добавляет к языку Java новое ключевое слово `strictfp`. С созданием Java 2, модель вычисления с плавающей точкой была слегка ослаблена, чтобы для некоторых процессоров, таких как Pentium, увеличить скорость вычислений с плавающими числами. Новая модель не требует усечения некоторых промежуточных значений, которые появляются во время вычислений. Изменяя класс или метод с помощью `strictfp`, вы гарантируете, что вычисления с плавающей точкой (и таким образом все усечения) выполняется точно так же, как в более ранних версиях Java. Усечение воздействует только на экспоненту некоторых операций. Когда некоторый класс модифицируется с помощью `strictfp`, все методы в этом классе автоматически модифицируются с помощью `strictfp`.

Например, следующий фрагмент сообщает Java, что нужно использовать первоначальную модель с плавающей точкой для вычислений во всех методах, определенных в `Myciass`:

```
strictfp class MyClass { //…
```

Большинство программистов никогда не использует `strictfp`, т. к. эта конструкция решает небольшой круг проблем.

1.7. Native-методы

Хотя это бывает редко и совершенно случайно, но иногда может возникнуть желание вызвать подпрограмму, которая написана на другом языке, а не на Java. Как правило, такая подпрограмма существует как выполняемый код для CPU и среды, в которой вы

работаете — то есть как "родной" (native) код. Например, нужно вызвать подпрограмму native-кода для достижения более быстрого времени выполнения. Или нужно использовать специализированную библиотеку типа статистического пакета. Однако из-за того, что Java-программы компилируются в байт-код, который затем интерпретируется (или компилируется "на лету") исполнительной системой Java, казалось бы, невозможно вызвать подпрограмму native-кода изнутри Java-программы. К счастью, это не так. В Java существует ключевое слово `native`, которое используется для объявления методов native-кода. После объявления эти методы можно вызывать внутри Java-программы точно так же, как вызывается любой другой метод Java.

Для объявления native-метода нужно предварить его заголовок модификатором `native`, при этом, однако, не следует определять никакого тела. Например:

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

После объявления native-метода, следует записать сам родной метод и выполнить довольно сложную процедуру для связи его с кодом Java.

Большинство родных методов записываются на `C`. Механизм, используемый для интеграции `C`-кода с Java-программой, называется JNI-интерфейсом (Java Native Interface) — native-интерфейс Java). Эта методология была создана для Java 1.1 и затем расширена и улучшена в Java 2. Следующее описание обеспечивает достаточную информацию для большинства приложений.

Лучше всего процесс воспринимается на примере. Для начала, введем следующую короткую программу, которая использует метод `native` с именем `test()`:

Программа 81. Связь с языком С

```
// Файл NativeDemo.java
// Простой пример, который использует native-метод.
public class NativeDemo {
   int i;
   public static void main(String args[]) {
      NativeDemo ob = new NativeDemo();
      ob.i = 10;
      System.out.println("Этот ob.i перед native-методом:" + ob.i);
      ob.test();                // Вызов native-метода
      System.out.println("Этот ob.i после native-метода:" + ob.i);
   }
// Объявить native-метод
   public native void test();
// Загрузить DLL, который содержит static-метод
   static {
      System.loadLibrary("NativeDemo");
   }
```

```
}
```

Заметим, что метод test() объявлен как native и не имеет тела. Он будет реализован на C. Обратите также внимание на блок static. Как объяснялось ранее, static-блок выполняется только один раз, когда программа начинает выполняться (или, более точно, когда его класс впервые загружается). В данном случае он используется для загрузки DLL-библиотеки (Dynamic Link Library — библиотека программ с динамической загрузкой.), которая содержит native-реализацию метода test(). (Далее вы увидите, как можно создать такую библиотеку.)

Библиотека загружается методом loadLibrary(), который является частью класса system. Вот его общая форма:

```
static void loadLibrary (String filename)
```

Здесь filename — строка, которая специфицирует имя файла, содержащего библиотеку. Для среды Windows 95/98/NT предполагается, что этот файл имеет расширение .dll.

После ввода программы, откомпилируйте ее, чтобы получить файл NativeDemo.class. Затем, вы должны использовать JDK-утилиту javah.exe для получения файла C/C++ заголовка NativeDemo.h. Файл NativeDemo.h нужно включить в реализацию метода test(). Для построения NativeDemo.h используйте следующую команду:

```
javah -jni NativeDemo
```

Данная команда производит файл заголовка с именем NativeDemo.h. Этот файл должен быть включен в C-файл, который реализует test(). Вывод указанной команды:

```
/* НЕ РЕДАКТИРУЙ ЭТОТ ФАЙЛ - он сгенерирован машиной */
#include <jni.h>
/* Заголовок класса NativeDemo */
#ifndef _Included_NativeDemo
#define _Included_NativeDemo
#ifdef __cplusplus
extern "C" {
#endif /*
* Class: NativeDemo
* Method:       test
* Signature:  ()V
*/
JNIEXPORT void JNI CALL Java_NativeDemo_test
      (JNIEnv *,  jobject);
#ifdef __cplusplus
}
#endif
#endif
```

Обратите особое внимание на следующую строку, определяющую прототип функции test(), которую вы будете создавать:

```
JNIEXPORT void JNI CALL Java_NativeDemo_test (JNIEnv *, jobject);
```

Заметим, что имя функции – `Java_NativeDemo_test()` - его нужно использовать как имя native-функции, которую вы реализуете. То есть вместо создания С-функции, названной `test()`, вы будете создавать функцию с именем `Java_NativeDemo_test()`. Компонент `NativeDemo` префикса добавляется потому, что он идентифицирует метод `test()` как часть класса `NativeDemo`. Помните, что другой класс может определить свой собственный native-метод `test()`, который полностью отличается от того, что объявлен в `NativeDemo`.

Включение в префикс имени класса обеспечивает возможность дифференцировать различные версии. Общее правило: native-функциям нужно давать имя, чей префикс включает имя класса, в котором они объявлены.

После создания необходимого файла заголовка вы можете написать свою реализацию `test()` и сохранить его в файле с именем `NativeDemo.c`:

```
/* Этот файл содержит С-версию метода test().  */
#include "NativeDemo.h"
#include <stdio.h>
JNIEXPORT void JNICALL Java_NativeDemo_test (JNIEnv *env, jobject obj) {
   jclass cls;
   jfieldID fid;
   jint i;
   printf("Запуск native-метода.\n");
   cls = (*env).GetObjectClass(obj);
   fid = (*env).GetFieldID(cls, "i", "I");
   if(fid == 0) {
      printf ("Невозможно получить id поля.\n");
      return;
   }
   i = (*env).GetIntField(obj, fid);
   printf("i = %d\n", i);
   (*env)->SetIntField(env, obj, fid, 2*i);
   printf("Завершение native-метода.\n");
}
```

Заметим, что файл включает `jni.h`, который содержит интерфейсную информацию. Этот файл обеспечивается Java-компилятором. Файл заголовка `NativeDemo.h` был создан ранее с помощью утилиты `javah`.

В этой функции метод `GetObjectClass()` используется для получения С-структуры, которая содержит информацию о классе `NativeDemo`. Метод `GetFieldID()` возвращает С-структуру с информацией о поле класса с именем "`i`". Метод `GetIntField()` извлекает первоначальное значение этого поля. Метод `SetIntField()` хранит обновленное значение в данном поле. (Дополнительные методы, которые обрабатывают другие типы данных, см. в файле `jni.h`.)

После создания NativeDemo.c нужно откомпилировать его и создать DLL-файл. Для этого используется Microsoft-компилятор C/C++ со следующей командной строкой:

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

Это создает файл с именем NativeDemo.dll. Как только указанная процедура проделана, можно выполнять Java-программу, которая сгенерирует следующий вывод:

```
Этот ob.i перед native-методом: 10
Запуск native-метода, i = 10
Завершение native-метода.
Этот ob.i после native-метода: 20
```

Специфика использования native зависит от реализации и среды. Кроме того, специфика способа взаимодействия с кодом Java подвергается изменениям. По деталям работы с native-методами нужно обращаться к документации системы разработки Java-программ.

Проблемы native-методов

Native-методы порождают большие надежды, потому что обеспечивают доступ к существующей базе библиотечных подпрограмм и более быструю работу во время выполнения. Но они порождают также две существенные проблемы:

□ **Потенциальный риск безопасности.** Поскольку native-метод выполняет фактический машинный код, он может получать доступ к любой части хост-системы. То есть native-код не ограничен средой выполнения Java. Это может привести к заражению вирусом, например. По этой же причине native-методы не могут использовать апплеты. Загрузка DLL-файлов может быть ограничена и подчинена одобрению руководителя службы безопасности.

□ **Потеря мобильности.** Поскольку native-код содержится в DLL-файле, он должен присутствовать на машине, выполняющей программу Java. Далее, так как любой native-метод зависит от CPU и операционной системы, каждый такой DLL-файл неизбежно непереносим. Таким образом, приложение Java, которое использует native-методы, будет способно выполниться только на машине, где был установлен соответствующий DLL-файл.

Использование native-методов должно быть ограничено, потому что они делают Java-программы непереносными и вносят существенный риск защиты.