



# Глава 10

## **Ввод-вывод данных**

## В этой главе...

- Потоки ввода-вывода
- Отличия байтовых и символьных потоков
- Классы для поддержки байтовых потоков
- Классы для поддержки символьных потоков
- Знакомство со встроенными потоками
- Использование байтовых потоков
- Использование байтовых потоков для файлового ввода-вывода
- Автоматическое закрытие файлов с помощью инструкции `try` с ресурсами
- Чтение и запись двоичных данных
- Манипулирование файлами с произвольным доступом
- Использование символьных потоков
- Использование символьных потоков для файлового ввода-вывода
- Применение оболочек типов Java для преобразования числовых строк

**В** предыдущих главах уже рассматривались примеры программ, в которых использовались отдельные элементы подсистемы ввода-вывода Java, в частности, метод `println()`, но все это делалось без каких-либо формальных пояснений. В Java подсистема ввода-вывода основана на иерархии классов, и поэтому ее невозможно было рассматривать, не узнав предварительно, что такое классы, наследование и исключения. Теперь, когда вы уже в достаточной степени к этому подготовлены, мы можем приступить к обсуждению средств ввода-вывода.

Следует отметить, что подсистема ввода-вывода Java очень обширна и включает множество классов, интерфейсов и методов. Отчасти это объясняется тем, что в Java определены фактически две полноценные подсистемы ввода-вывода: одна — для обмена байтами, другая — для обмена символами. Здесь нет возможности рассмотреть все аспекты ввода-вывода в Java, ведь для этого потребовалась бы отдельная книга. Поэтому в данной главе будут рассмотрены лишь наиболее важные и часто используемые языковые средства ввода-вывода. Правда, элементы подсистемы ввода-вывода в Java тесно взаимосвязаны, и поэтому, уяснив основы, вы легко освоите все остальные нюансы этой подсистемы.

Прежде чем приступить к рассмотрению подсистемы ввода-вывода, необходимо сделать следующее замечание. Классы, описанные в этой главе, предназначены для консольного и файлового ввода-вывода. Они не используются для

создания графических пользовательских интерфейсов. Поэтому при создании оконных приложений они вам не пригодятся. Для создания графических интерфейсов в Java предусмотрены другие средства. Они будут представлены в главах 16 и 17, где вы познакомитесь с библиотеками Swing и JavaFX соответственно.

## Потоковая организация ввода-вывода в Java

В Java операции ввода-вывода реализованы на основе потоков. Поток — это абстрактная сущность, представляющая устройства ввода-вывода, которая выдает и получает информацию. За связь потоков с физическими устройствами отвечает подсистема ввода-вывода, что позволяет работать с разными устройствами, используя одни и те же классы и методы. Например, методы вывода на консоль в равной степени могут быть использованы для записи данных в дисковый файл. Для реализации потоков используется иерархия классов, содержащихся в пакете `java.io`.

## Байтовые и символьные потоки

В современных версиях Java определены два типа потоков: байтовые и символьные. (Первоначально в Java были доступны только байтовые потоки, но вскоре были реализованы и символьные.) Байтовые потоки предоставляют удобные средства для управления вводом и выводом байтов. Например, их можно использовать для чтения и записи двоичных данных. Потоки этого типа особенно удобны при работе с файлами. С другой стороны, символьные потоки ориентированы на обмен символьными данными. В них применяется кодировка Unicode, и поэтому их легко интернационализировать. Кроме того, в некоторых случаях символьные потоки более эффективны по сравнению с байтовыми потоками.

Необходимость поддерживать два разных типа потоков ввода-вывода привела к созданию двух иерархий классов: одна — для байтовых, другая — для символьных данных. Из-за того что число классов достаточно велико, на первый взгляд подсистема ввода-вывода кажется сложнее, чем она есть на самом деле. Просто знайте, что в большинстве случаев функциональные возможности символьных потоков идентичны возможностям байтовых.

Вместе с тем на самом нижнем уровне все средства ввода-вывода имеют байтовую организацию. Символьные потоки лишь предоставляют удобные и эффективные средства, адаптированные к специфике обработки символов.

## Классы байтовых потоков

Байтовые потоки определены с использованием двух иерархий классов, на вершинах которых находятся абстрактные классы `InputStream` и `OutputStream` соответственно. В классе `InputStream` определены свойства, общие для байтовых потоков ввода, а в классе `OutputStream` — свойства, общие для байтовых потоков вывода.

Производными от классов `InputStream` и `OutputStream` являются конкретные подклассы, реализующие различные функциональные возможности и учитывающие особенности обмена данными с разными устройствами, например ввода-вывода в файлы на диске. Классы байтовых потоков перечислены в табл. 10.1. Пусть вас не пугает большое количество этих классов: изучив один из них, вы легко освоите остальные.

Таблица 10.1. Классы байтовых потоков

Класс байтового потока	Описание
<code>BufferedInputStream</code>	Буферизованный входной поток
<code>BufferedOutputStream</code>	Буферизованный выходной поток
<code>ByteArrayInputStream</code>	Входной поток для чтения из байтового массива
<code>ByteArrayOutputStream</code>	Выходной поток для записи в байтовый массив
<code>DataInputStream</code>	Входной поток, включающий методы для чтения стандартных типов данных Java
<code>DataOutputStream</code>	Выходной поток, включающий методы для записи стандартных типов данных Java
<code>FileInputStream</code>	Входной поток для чтения из файла
<code>FileOutputStream</code>	Выходной поток для записи в файл
<code>FilterInputStream</code>	Реализация класса <code>InputStream</code>
<code>FilterOutputStream</code>	Реализация класса <code>OutputStream</code>
<code>InputStream</code>	Абстрактный класс, описывающий потоковый ввод
<code>ObjectInputStream</code>	Входной поток для объектов
<code>ObjectOutputStream</code>	Выходной поток для объектов
<code>OutputStream</code>	Абстрактный класс, описывающий потоковый вывод
<code>PipedInputStream</code>	Входной канал
<code>PipedOutputStream</code>	Выходной канал
<code>PrintStream</code>	Выходной поток, включающий методы <code>print()</code> и <code>println()</code>
<code>PushbackInputStream</code>	Входной поток, позволяющий возвращать байты обратно в поток
<code>SequenceInputStream</code>	Входной поток, сочетающий в себе несколько потоков, которые читаются последовательно, один после другого

## Классы символьных потоков

Символьные потоки также определены с использованием двух иерархий классов, вершины которых на этот раз представлены абстрактными классами `Reader` и `Writer` соответственно. Класс `Reader` и его подклассы используются для чтения, а класс `Writer` и его подклассы — для записи данных. Конкретные

классы, производные от классов `Reader` и `Writer`, оперируют символами в кодировке `Unicode`.

Классы, производные от классов `Reader` и `Writer`, предназначены для выполнения различных операций ввода-вывода символов. В целом символьные классы представляют собой аналоги соответствующих классов, предназначенных для работы с байтовыми потоками. Классы символьных потоков перечислены в табл. 10.2.

**Таблица 10.2. Классы символьных потоков**

Класс символьного потока	Описание
<code>BufferedReader</code>	Буферизованный входной символьный поток
<code>BufferedWriter</code>	Буферизованный выходной символьный поток
<code>CharArrayReader</code>	Входной поток для чтения из символьного массива
<code>CharArrayWriter</code>	Выходной поток для записи в символьный массив
<code>FileReader</code>	Входной поток для чтения из файла
<code>FileWriter</code>	Выходной поток для записи в файл
<code>FilterReader</code>	Фильтрующий входной поток
<code>FilterWriter</code>	Фильтрующий выходной поток
<code>InputStreamReader</code>	Входной поток, транслирующий байты в символы
<code>LineNumberReader</code>	Входной поток, подсчитывающий строки
<code>OutputStreamWriter</code>	Выходной поток, транслирующий символы в байты
<code>PipedReader</code>	Входной канал
<code>PipedWriter</code>	Выходной канал
<code>PrintWriter</code>	Выходной поток, включающий методы <code>print()</code> и <code>println()</code>
<code>PushbackReader</code>	Входной поток, позволяющий возвращать символы обратно в поток
<code>Reader</code>	Абстрактный класс, описывающий символьный ввод
<code>StringReader</code>	Входной поток для чтения из строки
<code>StringWriter</code>	Выходной поток для записи в строку
<code>Writer</code>	Абстрактный класс, описывающий символьный вывод

## Встроенные потоки

Как вы уже знаете, во все программы на Java автоматически импортируется пакет `java.lang`, в котором определен класс `System`, инкапсулирующий некоторые свойства среды выполнения. Помимо прочего, в нем содержатся predefined переменные `in`, `out` и `err`, представляющие стандартные потоки ввода-вывода. Эти поля объявлены как `public`, `final` и `static`, т.е. к ним можно обращаться из любой другой части программы, не ссылаясь на конкретный объект типа `System`.

Переменная `System.out` ссылается на стандартный выходной поток, который по умолчанию связан с консолью. Переменная `System.in` ссылается на стандартный входной поток, который по умолчанию связан с клавиатурой. И наконец, переменная `System.err` ссылается на стандартный поток ошибок, который, как и выходной поток, также связан по умолчанию с консолью. При необходимости каждый из этих потоков может быть перенаправлен на любое другое устройство.

Поток `System.in` — это объект типа `InputStream`, а потоки `System.out` и `System.err` — объекты типа `PrintStream`. Все эти потоки — байтовые, хотя обычно они используются для чтения и записи символов с консоли и на консоль. Дело в том, что в первоначальной спецификации Java, в которой символьные потоки вообще отсутствовали, все предопределенные потоки были байтовыми. Как вы далее увидите, по мере необходимости их можно поместить в классы-оболочки символьных потоков.

## Использование байтовых потоков

Начнем рассмотрение подсистемы ввода-вывода в Java с байтовых потоков. Как уже отмечалось, на вершине иерархии байтовых потоков находятся классы `InputStream` и `OutputStream`. Методы класса `InputStream` перечислены в табл. 10.3, а методы класса `OutputStream` — в табл. 10.4. При возникновении ошибок во время выполнения методы классов `InputStream` и `OutputStream` могут генерировать исключения `IOException`. Определенные в этих двух абстрактных классах методы доступны во всех подклассах. Таким образом, они образуют минимальный набор функций ввода-вывода, общий для всех байтовых потоков.

Таблица 10.3. Методы, определенные в классе `InputStream`

Метод	Описание
<code>int available()</code>	Возвращает количество байтов ввода, доступных в данный момент для чтения
<code>void mark(int numBytes)</code>	Помещает в текущую позицию входного потока метку, которая будет находиться там до тех пор, пока не будет прочитано количество байтов, определяемое параметром <code>numBytes</code>
<code>boolean markSupported()</code>	Возвращает значение <code>true</code> , если методы <code>mark()</code> и <code>reset()</code> поддерживаются вызывающим потоком
<code>int read()</code>	Возвращает целочисленное представление следующего байта в потоке. По достижении конца потока возвращается значение <code>-1</code>

Окончание табл. 10.3

Метод	Описание
<code>int read(byte buffer[])</code>	Пытается прочитать <code>buffer.length</code> байтов в массив <code>buffer</code> , возвращая фактическое количество успешно прочитанных байтов. По достижении конца потока возвращается значение <code>-1</code>
<code>int read(byte buffer[], int offset, int numBytes)</code>	Пытается прочитать <code>buffer.length</code> байтов в массив <code>buffer</code> , начиная с элемента <code>buffer[offset]</code> , и возвращает фактическое количество успешно прочитанных байтов. По достижении конца потока возвращается значение <code>-1</code>
<code>byte[] readAllBytes()</code>	Считывает и возвращает в виде байтового массива все байты, доступные в потоке. В результате попытки считывания конца потока будет создан пустой массив (добавлено в JDK 9)
<code>int readNBytes(byte buffer[], int offset, int numBytes)</code>	Попытка считывания <code>numBytes</code> байтов в буфер <code>buffer</code> , начинающийся с <code>buffer[offset]</code> , возвращая количество успешно считанных байтов. В результате попытки считывания конца потока будет прочитано 0 байтов (добавлено в JDK 9)
<code>void reset()</code>	Сбрасывает входной указатель на ранее установленную метку
<code>long skip (long numBytes)</code>	Пропускает <code>numBytes</code> входных байтов, возвращая фактическое количество пропущенных байтов
<code>long transferTo(OutputStream outStrm)</code>	Копирует содержимое вызывающего потока в <code>outStrm</code> , возвращая количество скопированных байтов (добавлено в JDK 9)

Таблица 10.4. Методы, определенные в классе `OutputStream`

Метод	Описание
<code>void close()</code>	Закрывает выходной поток. Дальнейшие попытки записи будут генерировать исключение <code>IOException</code>
<code>void flush()</code>	Выполняет принудительную передачу содержимого выходного буфера в место назначения (тем самым очищая выходной буфер)
<code>void write(int b)</code>	Записывает один байт в выходной поток. Обратите внимание на то, что параметризует тип <code>int</code> , что позволяет вызывать

Метод	Описание
	метод <code>write()</code> с выражениями, не приводя их к типу <code>byte</code>
<code>void write(byte buffer[])</code>	Записывает полный массив байтов в выходной поток
<code>void write(byte buffer[], int offset, int numBytes)</code>	Записывает часть массива <code>buffer</code> в количестве <code>numBytes</code> байтов, начиная с элемента <code>buffer[offset]</code>

## Консольный ввод

Первоначально байтовые потоки были единственным средством, позволяющим выполнять консольный ввод, и во многих существующих программах на Java для этой цели по-прежнему используются исключительно байтовые потоки. Сейчас имеется возможность выбора между байтовыми и символьными потоками.

В коммерческом коде для чтения консольного ввода предпочтительнее использовать символьные потоки. Такой подход упрощает интернационализацию программ и облегчает их сопровождение. Ведь намного удобнее оперировать непосредственно символами, не тратя время и усилия на преобразование символов в байты и наоборот. Однако в простых служебных и прикладных программах, где данные, введенные с клавиатуры, обрабатываются непосредственно, удобно пользоваться байтовыми потоками. Именно по этой причине они здесь и рассматриваются.

Поток `System.in` является экземпляром класса `InputStream`, и благодаря этому обеспечивается автоматический доступ к методам, определенным в классе `InputStream`. К сожалению, для чтения байтов в классе `InputStream` определен только один метод ввода: `read()`. Ниже приведены три возможные формы объявления этого метода.

```
int read() throws IOException
int read(byte data[]) throws IOException
int read(byte data[], int start, int max) throws IOException
```

В главе 3 было продемонстрировано, как пользоваться первой формой метода `read()` для чтения отдельных символов с клавиатуры (а по сути, из потока стандартного ввода `System.in`). Достигнув конца потока, этот метод возвращает значение `-1`. Вторая форма метода `read()` предназначена для чтения данных из входного потока в массив `data`. Чтение завершается по достижении конца потока, при заполнении массива или возникновении ошибки. Метод возвращает количество прочитанных байтов или `-1`, если достигнут конец потока. И третья форма данного метода позволяет разместить прочитанные данные в массиве `data`, начиная с элемента, заданного с помощью индекса `start`. Максимальное количество байтов, которые могут быть введены в массив,



определяется параметром *max*. Метод возвращает число прочитанных байтов или значение  $-1$ , если достигнут конец потока. При возникновении ошибки в каждой из этих форм метода `read()` генерируется исключение `IOException`. Признаком конца потока ввода при чтении из `System.in` устанавливается после нажатия клавиши `<Enter>`.

Ниже приведен пример короткой программы, демонстрирующий чтение байтов из потока ввода `System.in` в массив. Следует иметь в виду, что исключения, которые могут быть сгенерированы при выполнении данной программы, обрабатываются за пределами метода `main()`. Подобный подход часто используется при чтении данных с консоли. По мере необходимости вы сможете самостоятельно организовать обработку ошибок.

```
// Чтение байтов с клавиатуры в массив
import java.io.*;

class ReadBytes {
    public static void main(String args[])
        throws IOException {
        byte data[] = new byte[10];

        System.out.println("Введите символы.");
        System.in.read(data); ← Чтение байтового массива с клавиатуры
        System.out.print("Вы ввели: ");
        for(int i=0; i < data.length; i++)
            System.out.print((char) data[i]);
    }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
Введите символы.
Read Bytes
Вы ввели: Read Bytes
```

## Вывод на консоль

Как и в случае консольного ввода, в Java для консольного вывода первоначально были предусмотрены только байтовые потоки. Символьные потоки были добавлены в версии Java 1.1. Для переносимого кода в большинстве случаев рекомендуется использовать символьные потоки. Однако, поскольку поток `System.out` — байтовый, он по-прежнему широко используется для побайтового вывода данных на консоль. Именно такой подход до сих пор применялся в примерах, представленных в книге. Поэтому существует необходимость рассмотреть его более подробно.

Вывод данных на консоль проще всего осуществлять с помощью уже знакомых вам методов `print()` и `println()`. Эти методы определены в классе `PrintStream` (на объект данного типа ссылается переменная потока

стандартного вывода `System.out`). Несмотря на то что `System.out` является байтовым потоком, его вполне можно использовать для простого консольного вывода.

Поскольку класс `PrintStream` является выходным потоком, производным от класса `OutputStream`, он также реализует низкоуровневый метод `write()`, который может быть использован для записи на консоль. Ниже приведена простейшая форма метода `write()`, определенного в классе `PrintStream`:

```
void write(int byteval)
```

Данный метод записывает в файл значение байта, переданное с помощью параметра `byteval`. Несмотря на то что этот параметр объявлен как `int`, в нем учитываются только младшие 8 бит. Ниже приведен простой пример программы, в которой метод `write()` используется для вывода символа X и символа перевода строки на консоль.

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

        b = 'X';
        System.out.write(b); ← Запись байта в поток
        System.out.write('\n');
    }
}
```

Вам не часто придется использовать метод `write()` для вывода на консоль, хотя в некоторых ситуациях он оказывается весьма кстати. Для этой цели намного удобнее пользоваться методами `print()` и `println()`.

В классе `PrintStream` реализованы два дополнительных метода, `printf()` и `format()`, которые позволяют управлять форматированием выводимых данных. Например, они позволяют указать для выводимых данных количество десятичных цифр, минимальную ширину поля или способ представления отрицательных числовых значений. И хотя эти методы не используются в примерах, представленных в данной книге, вам стоит обратить на них пристальное внимание, поскольку они могут пригодиться при написании прикладных программ.

## Чтение и запись файлов с использованием байтовых потоков

Язык Java предоставляет множество классов и методов, позволяющих читать и записывать данные из файлов и в файлы. Разумеется, чаще всего приходится обращаться к файлам, хранящимся на дисках. В Java все файлы имеют байтовую организацию, и поэтому для побайтового чтения и записи данных из файла и в файл предусмотрены соответствующие методы. Таким образом, файловые операции с использованием байтовых потоков довольно распространены.

Кроме того, для байтовых потоков ввода-вывода в файлы в Java допускается создавать оболочки в виде символьных объектов. Классы-оболочки будут рассмотрены далее.

Байтовые потоки, связанные с файлами, создаются с помощью классов `FileInputStream` или `FileOutputStream`. Чтобы открыть файл, достаточно создать объект одного из этих классов, передав конструктору имя файла в качестве параметра. Открытие файла необходимо для того, чтобы с ним можно было выполнять файловые операции чтения и записи.

## Чтение данных из файла

Файл открывается для чтения путем создания объекта типа `FileInputStream`. Для этой цели чаще всего используется следующая форма конструктора данного класса:

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

Имя файла, который требуется открыть, передается конструктору в параметре `имя_файла`. Если указанного файла не существует, генерируется исключение `FileNotFoundException`.

Для чтения данных из файла используется метод `read()`. Ниже приведена форма объявления этого метода, которой мы будем пользоваться в дальнейшем.

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

При каждом вызове метод `read()` читает байт из файла и возвращает его в виде целочисленного значения. По достижении конца файла этот метод возвращает значение `-1`. При возникновении ошибки метод генерирует исключение `IOException`. Как видите, в этой форме метод `read()` выполняет те же самые действия, что и одноименный метод, предназначенный для ввода данных с консоли.

После завершения операций с файлом следует закрыть его с помощью метода `close()`, имеющего следующую общую форму объявления:

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

При закрытии файла освобождаются связанные с ним системные ресурсы, которые вновь можно будет использовать для работы с другими файлами. Если этого не сделать, возможна *утечка памяти* из-за того, что часть памяти остается выделенной для ресурсов, которые больше не используются.

Ниже приведен пример программы, в которой метод `read()` используется для получения и отображения содержимого текстового файла. Имя файла указывается в командной строке при запуске программы. Обратите внимание на то, что ошибки ввода-вывода обрабатываются с помощью блока `try/catch`.

```
/* Отображение текстового файла.
```

```

При вызове этой программы следует указать имя файла,
содержимое которого требуется просмотреть.
Например, для вывода на экран содержимого файла TEST.TXT
необходимо ввести 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 нужно убедиться в том, что программе
        // передается имя файла
        if(args.length != 1) {
            System.out.println("Использование: ShowFile имя_файла");
            return;
        }

        try {
            fin = new FileInputStream(args[0]); ← Открытие файла
        } catch(FileNotFoundException exc) {
            System.out.println("Файл не найден");
            return;
        }

        try {
            // Чтение байтов, пока не встретится символ EOF
            do {
                i = fin.read(); ← Считывание данных из файла
                if(i != -1) System.out.print((char) i);
            } while(i != -1); ← Если i == -1, значит, достигнут конец файла
        } catch(IOException exc) {
            System.out.println("Ошибка при чтении файла");
        }

        try {
            fin.close(); ← Закрытие файла
        } catch(IOException exc) {
            System.out.println("Ошибка при закрытии файла");
        }
    }
}

```

Обратите внимание на то, что в этом примере файловый поток закрывается после завершения выполнения блока `try`, в котором осуществляется чтение данных. Такой способ не всегда оказывается удобным, и поэтому в Java имеется более совершенный и чаще используемый способ, предполагающий помещение вызова метода `close()` в блок `finally`. В этом случае все методы, получающие доступ к файлу, помещаются в блок `try`, а для закрытия файла используется блок `finally`. Благодаря этому файл закрывается независимо от того, как завершится блок `try`. Учитывая это, перепишем блок `try` из предыдущего примера в таком виде.

```

try {
    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);
} catch(IOException exc) {
    System.out.println("Ошибка при чтении файла");
    // Для закрытия файла используется блок finally
} finally {
    // Закрыть файл при выходе из блока try
    try {
        fin.close();
    } catch(IOException exc) {
        System.out.println("Ошибка при закрытии файла");
    }
}

```

Использование блока finally для закрытия файла

Данный способ обеспечивает, в частности, то преимущество, что в случае аварийного завершения программы из-за возникновения исключения, не связанного с операциями ввода-вывода, файл все равно будет закрываться в блоке `finally`. И если с аварийным завершением простых программ в связи с непредвиденными исключениями, как в большинстве примеров книги, еще можно как-то мириться, то в больших программах подобная ситуация вряд ли может считаться допустимой. Использование блока `finally` позволяет справиться с этой проблемой.

Иногда части программы, ответственные за открытие файла и осуществление доступа к нему, удобнее поместить в один блок `try` (не разделяя их), а для закрытия файла использовать блок `finally`. В качестве примера ниже приведена измененная версия рассмотренной выше программы `ShowFile`.

```

/* В этой версии программы те ее части, которые отвечают
   за открытие файла и получение доступа к нему, помещены
   в один блок try. Файл закрывается в блоке finally.
*/
import java.io.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;
        FileInputStream fin = null;
        // Сначала нужно убедиться в том, что программе
        // передается имя файла
        if(args.length != 1) {
            System.out.println("Использование: ShowFile имя_файла");
            return;
        }
    }
}

```

Переменная `fin` инициализируется значением `null`

```

// Открытие файла, чтение из него символов, пока
// не встретится признак конца файла EOF, и
// последующее закрытие файла в блоке finally
try {
    fin = new FileInputStream(args[0]);

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

} catch(FileNotFoundException exc) {
    System.out.println("Файл не найден.");
} catch(IOException exc) {
    System.out.println("Ошибка ввода-вывода");
} finally {
    // Файл закрывается в любом случае
    try {
        if(fin != null) fin.close(); ← Закрыть файл, если значение fin
    } catch(IOException exc) { ← не равно null
        System.out.println("Ошибка при закрытии файла");
    }
}
}
}

```

Обратите внимание на то, что переменная `fin` инициализируется значением `null`. В блоке `finally` файл закрывается только в том случае, если значение переменной `fin` не равно `null`. Это будет работать, поскольку переменная `fin` не содержит значение `null` лишь в том случае, когда файл был успешно открыт. Следовательно, если во время открытия файла возникнет исключение, метод `close()` не будет вызываться.

В этом примере блок `try/catch` можно сделать несколько более компактным. Поскольку исключение `FileNotFoundException` является подклассом исключения `IOException`, его не нужно перехватывать отдельно. В качестве примера ниже приведен блок `catch`, которым можно воспользоваться для перехвата обоих типов исключений, избегая независимого перехвата исключения `FileNotFoundException`. В данном случае выводится стандартное сообщение о возникшем исключении с описанием ошибки.

```

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

```

## СПРОСИМ У ЭКСПЕРТА

**ВОПРОС.** Метод `read()` возвращает значение `-1` по достижении конца файла, но для ошибки при попытке доступа к файлу специальное возвращаемое значение не предусмотрено. Почему?

**ОТВЕТ.** В Java для обработки ошибок используются исключения. Поэтому если метод `read()` или любой другой возвращает конкретное значение, то это автоматически означает, что в процессе его работы ошибка не возникла. Подобный подход многие считают гораздо более удобным, чем использование специальных кодов ошибок.

При таком подходе любая ошибка, в том числе и ошибка открытия файла, будет обработана единственной инструкцией `catch`. Благодаря своей компактности в большинстве примеров ввода-вывода, представленных в этой книге, используется именно такой способ. Следует, однако, иметь в виду, что он может оказаться не вполне пригодным в тех случаях, когда требуется отдельно обрабатывать ошибку открытия файла, вызванную, например, опечаткой при вводе имени файла. В подобных случаях рекомендуется сначала предложить пользователю заново ввести имя файла, а не входить сразу же в блок `try`, в котором осуществляется доступ к файлу.

## Запись в файл

Чтобы открыть файл для записи, следует создать объект типа `FileOutputStream`. Ниже приведены две наиболее часто используемые формы конструктора этого класса.

```
FileOutputStream(String имя_файла) throws FileNotFoundException
FileOutputStream(String имя_файла, boolean append)
    throws FileNotFoundException
```

В случае невозможности создания файла возникает исключение `FileNotFoundException`. Если файл с указанным именем уже существует, то в тех случаях, когда используется первая форма конструктора, этот файл удаляется. Вторая форма отличается от первой наличием параметра `append`. Если этот параметр имеет значение `true`, то записываемые данные добавляются в конец файла. В противном случае прежние данные в файле перезаписываются новыми.

Для записи данных в файл вызывается метод `write()`. Наиболее простая форма объявления этого метода выглядит так:

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

Данный метод записывает в поток байтовое значение, передаваемое с помощью параметра `byteval`. Несмотря на то что этот параметр объявлен как `int`, учитываются только младшие 8 бит его значения. Если в процессе записи возникает ошибка, генерируется исключение `IOException`.

По завершении работы с файлом его нужно закрыть с помощью метода `close()`, представленного ниже:

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

При закрытии файла освобождаются связанные с ним системные ресурсы, что позволяет использовать их в дальнейшем для работы с другими файлами. Кроме того, процедура закрытия файла гарантирует, что оставшиеся в буфере данные будут записаны на диск.

В следующем примере осуществляется копирование текстового файла. Имена исходного и целевого файлов указываются в командной строке.

```
/* Копирование текстового файла.
```

При вызове этой программы следует указать имена исходного и целевого файлов. Например, для копирования файла `FIRST.TXT` в файл `SECOND.TXT` в командной строке нужно ввести 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 нужно убедиться в том, что программе
        // передаются имена обоих файлов
        if(args.length != 2) {
            System.out.println("Использование: CopyFile -
                               источник и назначение");
            return;
        }

        // Копирование файла
        try {
            // Попытка открытия файлов
            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 exc) {
            System.out.println("Ошибка ввода-вывода: " + exc);
        } finally {
```



Чтение байтов  
из одного файла  
и запись их в другой  
файл



```

try {
    if(fin != null) fin.close();
} catch(IOException exc) {
    System.out.println("Ошибка при закрытии
                        входного файла");
}
try {
    if(fout != null) fout.close();
} catch(IOException exc) {
    System.out.println("Ошибка при закрытии выходного файла");
}
}
}
}

```

## Автоматическое закрытие файлов

В примерах программ, представленных в предыдущем разделе, для закрытия файлов, которые больше не нужны, метод `close()` вызывался явным образом. Такой способ закрытия файлов используется еще с тех пор, как вышла первая версия Java. Именно поэтому он часто встречается в существующих программах. Более того, он до сих пор остается вполне оправданным и полезным. Однако в версию JDK 7 включено новое средство, предоставляющее другой, более рациональный способ управления ресурсами, в том числе и потоками файлового ввода-вывода, автоматизирующий процесс закрытия файлов. Этот способ называемый *автоматическим управлением ресурсами*, основывается на новой разновидности инструкции `try`, называемой инструкцией *try с ресурсами*. Главное преимущество инструкции `try с ресурсами` заключается в том, что она предотвращает ситуации, в которых файл (или другой ресурс) непреднамеренно остается неосвобожденным даже после того, как необходимость в его использовании отпала. Как пояснялось ранее, если не позаботиться о своевременном закрытии файлов, это может привести к утечке памяти и прочим осложнениям в работе программы.

Так выглядит общая форма инструкции `try с ресурсами`:

```

try (описание_ресурса) {
    // использовать ресурс
}

```

Здесь *описание\_ресурса* включает в себя объявление и инициализацию ресурса, такого как файл. По сути, в это описание входит объявление переменной, которая инициализируется ссылкой на объект управляемого ресурса. По завершении блока `try` объявленный ресурс автоматически освобождается. Если этим ресурсом является файл, то он автоматически закрывается, что избавляет от необходимости вызывать метод `close()` явным образом. Инструкция `try с ресурсами` также может включать блоки `catch` и `finally`.

### Примечание

Начиная с JDK 9 спецификация ресурса `try` может состоять из переменной, которая была ранее объявлена и инициализирована в программе. Но эта переменная фактически должна быть последней в программе, чтобы ей не присваивалось новое значение после инициализации.

Область применимости таких инструкций `try` ограничена ресурсами, которые реализуют интерфейс `AutoCloseable`, определенный в пакете `java.lang`. В этом интерфейсе определен метод `close()`. Интерфейс `AutoCloseable` наследуется интерфейсом `Closeable`, определенным в пакете `java.io`. Оба интерфейса реализуются классами потоков, в том числе `FileInputStream` и `FileOutputStream`. Следовательно, инструкция `try` с ресурсами может применяться вместе с потоками, включая потоки файлового ввода-вывода.

В качестве примера ниже приведена переработанная версия программы `ShowFile`, в которой инструкция `try` с ресурсами используется для автоматического закрытия файла.

```

/* В этой версии программы ShowFile инструкция try с ресурсами
   применяется для автоматического закрытия файла, когда в нем
   больше нет необходимости.
*/

import java.io.*;

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

        // Прежде всего необходимо убедиться в том, что программе
        // передаются имена обоих файлов
        if(args.length != 1) {
            System.out.println("Использование: ShowFile имя_файла");
            return;
        }

        // Использование инструкции try с ресурсами для
        // открытия файла с последующим его закрытием после
        // того, как будет покинут блок try
        try(FileInputStream fin = new FileInputStream(args[0])) {
            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);
        } catch(IOException exc) {
            System.out.println("Ошибка ввода-вывода: " + exc);
        }
    }
}

```

← Блок try с ресурсами

Обратите внимание на то, как открывается файл в инструкции `try` с ресурсами:

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

Здесь сначала объявляется переменная `fin` типа `FileInputStream`, а затем этой переменной присваивается ссылка на файл, который выступает в качестве объекта, открываемого с помощью конструктора класса `FileInputStream`. Таким образом, в данной версии программы переменная `fin` является локальной по отношению к блоку `try` и создается при входе в этот блок. При выходе из блока `try` файл, связанный с переменной `fin`, автоматически закрывается с помощью неявно вызываемого метода `close()`. Это означает, что теперь отсутствует риск того, что вы забудете закрыть файл путем явного вызова метода `close()`. В этом и состоит главное преимущество автоматического управления ресурсами.

Важно понимать, что ресурс, объявленный в инструкции `try`, неявно имеет модификатор `final`. Это означает, что после создания ресурсной переменной ее значение не может быть изменено. Кроме того, ее область действия ограничивается блоком `try`.

С помощью одной подобной инструкции `try` можно управлять несколькими ресурсами. Для этого достаточно указать список объявлений ресурсов, разделенных точкой с запятой. В качестве примера ниже приведена переработанная версия программы `CopyFile`. В этой версии оба ресурса, `fin` и `fout`, управляются одной инструкцией `try`.

```
/* Версия программы CopyFile, в которой используется инструкция
   try с ресурсами. В ней демонстрируется управление двумя
   ресурсами (в данном случае – файлами) с помощью единственной
   инструкции try.
*/

import java.io.*;

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

        // Прежде всего необходимо убедиться в том, что программе
        // передаются имена обоих файлов
        if(args.length != 2) {
            System.out.println("Использование: CopyFile -
                               источник и назначение ");
            return;
        }

        // Открытие двух файлов и управление
        // ими с помощью инструкции 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 exc) {
        System.out.println("Ошибка ввода-вывода: " + exc);
    }
}

```

Управление  
двумя ресурсами

Обратите внимание на то, как входной и выходной файлы открываются в инструкции `try`.

```

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

```

По завершении этого блока `try` оба файла, на которые ссылаются переменные `fin` и `fout`, будут автоматически закрыты. Если сравнить эту версию программы с предыдущей версией, то можно заметить, что ее исходный код намного компактнее. Возможность создания более компактного кода является еще одним, дополнительным преимуществом инструкции `try` с ресурсами.

Стоит упомянуть еще об одной особенности инструкции `try` с ресурсами. Вообще говоря, возникшее при выполнении блока `try` исключение может породить другое исключение при закрытии ресурса в блоке `finally`. В случае “обычной” инструкции `try` первоначальное исключение теряется, будучи прерванным вторым исключением. Но в случае инструкции `try` с ресурсами второе исключение *подавляется*. При этом оно не теряется, а просто добавляется в список подавленных исключений, связанных с первым исключением. Этот список можно получить, вызвав метод `getSuppressed()`, определенный в классе `Throwable`.

Благодаря своим преимуществам инструкция `try` с ресурсами будет использоваться во многих оставшихся примерах программ в книге. Однако не менее важным остается и умение использовать рассмотренный ранее традиционный способ освобождения ресурсов с помощью явного вызова метода `close()`. И на то имеется ряд веских причин. Во-первых, среди уже существующих и повсеместно эксплуатируемых программ на Java немало таких, в которых применяется традиционный способ управления ресурсами. Поэтому нужно как следует усвоить традиционный подход и уметь использовать его для сопровождения устаревшего кода. Во-вторых, переход к использованию версии JDK 7 или выше может произойти не сразу, а следовательно, придется работать с предыдущей версией данного комплекта. В этом случае воспользоваться преимуществами инструкции `try` с ресурсами не удастся и нужно будет применять

традиционный способ управления ресурсами. И наконец, в некоторых случаях закрытие ресурса явным образом оказывается более эффективным, чем его автоматическое освобождение. И все же, если вы работаете с современными версиями Java, вариант автоматического управления ресурсами, как более рациональный и надежный, следует считать предпочтительным.

## Чтение и запись двоичных данных

В приведенных до сих пор примерах программ читались и записывались байтовые значения, содержащие символы в коде ASCII. Но аналогичным образом можно организовать чтение и запись любых типов данных. Допустим, требуется создать файл, содержащий значения типа `int`, `double` или `short`. Для чтения и записи простых типов данных в Java предусмотрены классы `DataInputStream` и `DataOutputStream`.

Класс `DataOutputStream` реализует интерфейс `DataOutput`, в котором определены методы, позволяющие записывать в файл значения любых примитивных типов. Следует, однако, иметь в виду, что данные записываются во внутреннем двоичном формате, а не в виде последовательности символов. Методы, наиболее часто применяемые для записи простых типов данных в Java, приведены в табл. 10.5. При возникновении ошибки ввода-вывода каждый из них может генерировать исключение `IOException`.

**Таблица 10.5. Наиболее часто используемые методы вывода данных, определенные в классе `DataOutputStream`**

Метод	Описание
<code>void writeBoolean(boolean val)</code>	Записывает логическое значение, определяемое параметром <code>val</code>
<code>void writeByte(int val)</code>	Записывает младший байт целочисленного значения, определяемого параметром <code>val</code>
<code>void writeChar(int val)</code>	Записывает значение, определяемое параметром <code>val</code> , интерпретируя его как символ
<code>void writeDouble(double val)</code>	Записывает значение типа <code>double</code> , определяемое параметром <code>val</code>
<code>void writeFloat(float val)</code>	Записывает значение типа <code>float</code> , определяемое параметром <code>val</code>
<code>void writeInt(int val)</code>	Записывает значение типа <code>int</code> , определяемое параметром <code>val</code>
<code>void writeLong(long val)</code>	Записывает значение типа <code>long</code> , определяемое параметром <code>val</code>
<code>void writeShort(int val)</code>	Записывает целочисленное значение, определяемое параметром <code>val</code> , преобразуя его в тип <code>short</code>

Ниже приведен конструктор класса `DataOutputStream`. Обратите внимание на то, что при вызове ему передается экземпляр класса `OutputStream`.

```
DataOutputStream(OutputStream outputStream)
```

Здесь `outputStream` — выходной поток, в который записываются данные. Для того чтобы организовать запись данных в файл, следует передать конструктору в качестве параметра `outputStream` объект типа `FileOutputStream`.

Класс `DataInputStream` реализует интерфейс `DataInput`, предоставляющий методы для чтения всех примитивных типов данных Java (табл. 10.6). При возникновении ошибки ввода-вывода каждый из них может генерировать исключение `IOException`. Класс `DataInputStream` построен на основе экземпляра класса `InputStream`, перекрывая его методами для чтения различных типов данных Java. Однако в потоке типа `DataInputStream` данные читаются в двоичном виде, а не в удобной для чтения форме. Ниже приведен конструктор класса `DataInputStream`:

```
DataInputStream(InputStream inputStream)
```

Здесь `inputStream` — это поток, связанный с создаваемым экземпляром класса `DataInputStream`. Для того чтобы организовать чтение данных из файла, следует передать конструктору в качестве параметра `inputStream` объект типа `FileInputStream`.

**Таблица 10.6. Наиболее часто используемые методы ввода данных, определенные в классе `DataInputStream`**

Метод	Описание
<code>boolean readBoolean()</code>	Читает значение типа <code>boolean</code>
<code>byte readByte()</code>	Читает значение типа <code>byte</code>
<code>char readChar()</code>	Читает значение типа <code>char</code>
<code>double readDouble()</code>	Читает значение типа <code>double</code>
<code>float readFloat()</code>	Читает значение типа <code>float</code>
<code>int readInt()</code>	Читает значение типа <code>int</code>
<code>long readLong()</code>	Читает значение типа <code>long</code>
<code>short readShort()</code>	Читает значение типа <code>short</code>

Ниже приведен пример программы, демонстрирующий использование классов `DataOutputStream` и `DataInputStream`. В этой программе данные разных типов сначала записываются в файл, а затем читаются из него.

```
// Запись и чтение двоичных данных

import java.io.*;

class RWData {
    public static void main(String args[])
```

```

{
    int i = 10;
    double d = 1023.56;
    boolean b = true;

    // Запись ряда значений
    try (DataOutputStream dataOut =
        new DataOutputStream(new FileOutputStream("testdata")))
    {
        System.out.println("Записано: " + i);
        dataOut.writeInt(i); ←
        System.out.println("Записано: " + d);
        dataOut.writeDouble(d); ←
        System.out.println("Записано: " + b);
        dataOut.writeBoolean(b); ←
        System.out.println("Записано: " + 12.2 * 7.4);
        dataOut.writeDouble(12.2 * 7.4); ←
    }
    catch (IOException exc) {
        System.out.println("Ошибка при записи");
        return;
    }

    System.out.println();

    // А теперь прочитать записанные значения
    try (DataInputStream dataIn =
        new DataInputStream(new FileInputStream("testdata")))
    {
        i = dataIn.readInt(); ←
        System.out.println("Прочитано: " + i);
        d = dataIn.readDouble(); ←
        System.out.println("Прочитано: " + d);
        b = dataIn.readBoolean(); ←
        System.out.println("Прочитано: " + b);
        d = dataIn.readDouble(); ←
        System.out.println("Прочитано: " + d);
    }
    catch (IOException exc) {
        System.out.println("Ошибка при чтении");
    }
}
}

```

Запись двоичных данных

Считывание двоичных данных

В результате выполнения этой программы получим следующий результат.

Записано: 10  
 Записано: 1023.56  
 Записано: true  
 Записано: 90.28

Прочитано: 10  
 Прочитано: 1023.56  
 Прочитано: true  
 Прочитано: 90.28

### Упражнение 10.1 Утилита сравнения файлов

`CompFiles.java` В этом упражнении нам предстоит создать простую, но очень полезную утилиту для сравнения содержимого файлов. В ходе выполнения этой служебной программы сначала открываются два сравниваемых файла, а затем данные читаются из них и сравниваются по соответствующему количеству байтов. Если на какой-то стадии операция сравнения дает отрицательный результат, это означает, что содержимое обоих файлов не одинаково. Если же конец обоих файлов достигается одновременно, это означает, что они содержат одинаковые данные. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте файл `CompFiles.java`.
2. Введите в файл `CompFiles.java` приведенный ниже исходный код.

```
/*
   Упражнение 10.1

   Сравнение двух файлов.

   При вызове этой программы следует указать имена
   сравниваемых файлов. Например, чтобы сравнить файл
   FIRST.TXT с файлом SECOND.TXT, в командной строке
   нужно ввести следующую команду:

   java CompFile FIRST.TXT SECOND.TXT
*/

import java.io.*;

class CompFiles {
    public static void main(String args[])
    {
        int i=0, j=0;

        // Прежде всего необходимо убедиться в том, что программе
        // передаются имена обоих файлов
        if(args.length !=2 ) {
```



```

        System.out.println("Использование: CompFiles файл1 файл2");
        return;
    }

    // Сравнение файлов
    try (FileInputStream f1 = new FileInputStream(args[0]);
        FileInputStream f2 = new FileInputStream(args[1]))
    {
        // Проверка содержимого каждого файла
        do {
            i = f1.read();
            j = f2.read();
            if(i != j) break;
        } while(i != -1 && j != -1);

        if(i != j)
            System.out.println("Содержимое файлов отличается");
        else
            System.out.println("Содержимое файлов совпадает");
    } catch(IOException exc) {
        System.out.println("Ошибка ввода-вывода: " + exc);
    }
}
}

```

3. Перед запуском программы скопируйте файл `CompFiles.java` во временный файл `temp`, а затем введите в командной строке такую команду:

```
java CompFiles CompFiles.java temp
```

Программа сообщит, что файлы имеют одинаковое содержимое. Далее сравните файл `CompFiles.java` с рассмотренным ранее файлом `CopyFile.java`, введя в командной строке следующую команду:

```
java CompFiles CompFiles.java CopyFile.java
```

Эти файлы имеют различное содержимое, о чем и сообщит программа `CompFiles`.

4. Попробуйте самостоятельно включить в программу `CompFiles` дополнительные возможности. В частности, предусмотрите возможность выполнять сравнение без учета регистра символов. Кроме того, программу `CompFiles` можно доработать так, чтобы она выводила номер позиции, в которой находится первая пара отличающихся символов.

## Файлы с произвольным доступом

До сих пор мы имели дело с последовательными файлами, содержимое которых вводилось и выводилось побайтово, т.е. строго по порядку. Но в Java предоставляется также возможность обращаться к хранящимся в файле данным в произвольном порядке. Для этой цели предусмотрен класс `RandomAccessFile`,

инкапсулирующий файл с произвольным доступом. Класс `RandomAccessFile` не является производным от класса `InputStream` или `OutputStream`. Вместо этого он реализует интерфейсы `DataInput` и `DataOutput`, в которых объявлены основные методы ввода-вывода. Кроме того, он поддерживает запросы с позиционированием, т.е. позволяет задавать положение указателя файла произвольным образом. Ниже приведен конструктор класса `RandomAccessFile`, который мы будем использовать далее.

```
RandomAccessFile(String имя_файла, String доступ)
    throws FileNotFoundException
```

Здесь конкретный файл указывается с помощью параметра *имя\_файла*, а параметр *доступ* определяет, какой именно тип доступа будет использоваться для обращения к файлу. Если параметр *доступ* принимает значение "r", то данные могут читаться из файла, но не записываться в него. Если же указан тип доступа "rw", то файл открывается как для чтения, так и для записи. Параметр *доступ* также может принимать значения "rws" и "rwd" (для локальных устройств), которые определяют немедленное сохранение файла на физическом устройстве.

Метод `seek()`, общая форма объявления которого приведена ниже, предназначен для установки текущего положения указателя файла.

```
void seek(long новая_позиция) throws IOException
```

Здесь параметр *новая\_позиция* определяет новое положение указателя файла в байтах относительно начала файла. Операция чтения или записи, следующая после вызова метода `seek()`, будет выполняться относительно нового положения указателя.

В классе `RandomAccessFile` определены методы `read()` и `write()`. Этот класс реализует также интерфейсы `DataInput` и `DataOutput`, т.е. в нем доступны методы чтения и записи простых типов, например `readInt()` и `writeDouble()`.

Ниже приведен пример программы, демонстрирующий ввод-вывод с произвольным доступом. В этой программе шесть значений типа `double` сначала записываются в файл, а затем читаются из него, причем порядок их чтения отличается от порядка записи.

```
// Демонстрация произвольного доступа к файлам
```

```
import java.io.*;
```

```
class RandomAccessDemo {
    public static void main(String args[])
    {
        double data[] = { 19.4, 10.1, 123.54, 33.0, 87.9, 74.25 };
        double d;
```

```
        // Открыть и использовать файл с произвольным доступом
```

```
        try (RandomAccessFile raf =
```

```
            new RandomAccessFile("random.dat", "rw"))
```

← Открыть файл с произвольным доступом

```

{
    // Запись значения в файл
    for(int i=0; i < data.length; i++) {
        raf.writeDouble(data[i]);
    }

    // Считывание отдельных значений из файла
    raf.seek(0); // найти первое значение типа double
    d = raf.readDouble();
    System.out.println("Первое значение: " + d);

    raf.seek(8) ; // найти второе значение типа double ← Установка
    d = raf.readDouble(); // указателя
    System.out.println("Второе значение: " + d); // на файл
                                                    // с помощью
                                                    // метода seek()

    raf.seek(8 * 3); // найти четвертое значение типа double
    d = raf.readDouble();
    System.out.println("Четвертое значение: " + d);

    System.out.println();

    // Прочитать значения через одно
    System.out.println("Чтение значений с нечетными
        порядковыми номерами: ");
    for(int i=0; i < data.length; i+=2) {
        raf.seek(8 * i); // найти i-е значение типа double
        d = raf.readDouble();
        System.out.print(d + " ");
    }
}
catch(IOException exc) {
    System.out.println("Ошибка ввода-вывода: " + exc);
}
}
}

```

Результат выполнения данной программы выглядит следующим образом.

```

Первое значение: 19.4
Первое значение: 10.1
Четвертое значение 33.0

```

```

Чтение значений с нечетными порядковыми номерами:
19.4 123.54 87.9

```

Отдельное замечание следует сделать относительно позиций расположения значений в файле. Поскольку для хранения значения типа `double` требуется 8 байтов, каждое последующее значение начинается на 8 байтовой границе предыдущего значения. Иными словами, первое числовое значение начинается с нулевого байта, второе — с 8-го байта, третье — с 16-го и т.д. Поэтому для чтения четвертого значения указатель файла должен быть установлен при вызове метода `seek()` на позиции 24-го байта.

## СПРОСИМ У ЭКСПЕРТА

**ВОПРОС.** В документации по JDK упоминается класс `Console`. Можно ли воспользоваться им для обмена данными с консолью?

**ОТВЕТ.** Да, можно. Класс `Console` впервые появился в версии Java 6 и служит для ввода-вывода данных на консоль. Этот класс создан в основном для удобства работы с консолью, поскольку большая часть его функциональных возможностей доступна в стандартных потоках ввода-вывода `System.in` и `System.out`. Но пренебрегать классом `Console` все же не следует, ведь с его помощью можно упростить некоторые типы операций с консолью, в том числе чтение символьных строк, вводимых с клавиатуры.

Класс `Console` не предоставляет конструкторы. Для получения объекта данного типа следует вызывать статический метод `System.console()`, который также был включен в версию Java 6. Ниже приведена общая форма объявления этого метода:

```
static Console console()
```

Если консоль доступна, то этот метод возвращает ссылку на соответствующий объект. В противном случае возвращается пустое значение `null`. Консоль доступна не всегда: обращение к ней запрещено, если программа выполняется в фоновом режиме.

В классе `Console` определен ряд методов, поддерживающих ввод-вывод, например `readLine()` и `printf()`. В нем также содержится метод `readPassword()`, предназначенный для получения пароля. При вводе пароля с клавиатуры его символы отображаются на экране с помощью замещающих знаков, не раскрывая пароль. С помощью средств класса `Console` можно также получить ссылки на объекты типа `Reader` и `Writer`, связанные с консолью. Таким образом, класс `Console` может оказаться очень полезным при написании некоторых видов приложений.

## Использование символьных потоков Java

Как говорилось в предыдущих разделах, байтовые потоки Java отличаются эффективностью и удобством использования. Но во всем, что касается ввода-вывода символов, они далеки от идеала. Для преодоления этого недостатка в Java определены классы символьных потоков. На вершине иерархии классов, поддерживающих символьные потоки, находятся абстрактные классы `Reader` и `Writer`. Методы класса `Reader` приведены в табл. 10.7, а методы класса `Writer` — в табл. 10.8. В большинстве этих методов может генерироваться исключение `IOException`. Методы, определенные в этих абстрактных классах, доступны во всех их подклассах. В совокупности эти методы предоставляют минимальный набор функций ввода-вывода, которые будут иметь все символьные потоки.

Таблица 10.7. Методы, определенные в классе `Reader`

Метод	Описание
<code>abstract void close()</code>	Закрывает источник ввода. Дальнейшие попытки чтения будут генерировать исключение <code>IOException</code>
<code>void mark (int numChars)</code>	Помещает в текущую позицию входного потока метку, которая будет находиться там до тех пор, пока не будет прочитано количество байтов, определяемое параметром <code>numChars</code>
<code>boolean markSupported()</code>	Возвращает значение <code>true</code> , если методы <code>mark()</code> и <code>reset()</code> поддерживаются вызывающим потоком
<code>int read()</code>	Возвращает целочисленное представление следующего символа в вызывающем входном потоке. По достижении конца потока возвращается значение <code>-1</code>
<code>int read(char buffer[])</code>	Пытается прочитать <code>buffer.length</code> символов в массив <code>buffer</code> , возвращая фактическое количество успешно прочитанных символов. По достижении конца потока возвращается значение <code>-1</code>
<code>abstract int read(char buffer[], int offset, int numChars)</code>	Пытается прочитать количество символов, определяемое параметром <code>numChars</code> , в массив <code>buffer</code> , начиная с элемента <code>buffer[offset]</code> . По достижении конца потока возвращается значение <code>-1</code>
<code>int read(CharBuffer buffer)</code>	Пытается заполнить буфер, определяемый параметром <code>buffer</code> , и возвращает количество успешно прочитанных символов. По достижении конца потока возвращается значение <code>-1</code> . <code>CharBuffer</code> — это класс, инкапсулирующий последовательность символов, например строку
<code>boolean ready()</code>	Возвращает значение <code>true</code> , если следующий запрос на получение символа может быть выполнен без ожидания. В противном случае возвращается значение <code>false</code>
<code>void reset()</code>	Сбрасывает входной указатель на ранее установленную метку
<code>long skip(long numChars)</code>	Пропускает <code>numChars</code> символов во входном потоке, возвращая фактическое количество пропущенных символов

Таблица 10.8. Методы, определенные в классе `Writer`

Метод	Описание
<code>Writer append(char ch)</code>	Добавляет символ <i>ch</i> в конец вызывающего выходного потока, возвращая ссылку на вызывающий поток
<code>Writer append(CharSequence chars)</code>	Добавляет последовательность символов <i>chars</i> в конец вызывающего потока, возвращая ссылку на вызывающий поток. <code>CharSequence</code> — это интерфейс, определяющий операции над последовательностями символов, выполняемые в режиме “только чтение”
<code>Writer append(CharSequence chars, int begin, int end)</code>	Добавляет последовательность символов <i>chars</i> в конец текущего потока, начиная с позиции, определяемой параметром <i>begin</i> , и заканчивая позицией, определяемой параметром <i>end</i> . Возвращает ссылку на вызывающий поток. <code>CharSequence</code> — это интерфейс, определяющий операции над последовательностями символов, выполняемые в режиме “только чтение”
<code>abstract void close()</code>	Закрывает выходной поток. Дальнейшие попытки чтения будут генерировать исключение <code>IOException</code>
<code>abstract void flush()</code>	Выполняет принудительную передачу содержимого выходного буфера в место назначения (тем самым очищая выходной буфер)
<code>void write(int ch)</code>	Записывает один символ в вызывающий выходной поток. Обратите внимание на то, что параметр имеет тип <code>int</code> , что позволяет вызывать метод <code>write()</code> с выражениями, не приводя их к типу <code>char</code>
<code>void write(char buffer[])</code>	Записывает полный массив символов <i>buffer</i> в вызывающий выходной поток
<code>abstract void write(char buffer[], int offset, int numChars)</code>	Записывает часть массива символов <i>buffer</i> в количестве <i>numChars</i> символов, начиная с элемента <i>buffer[offset]</i> , в вызывающий выходной поток
<code>void write(String str)</code>	Записывает строку <i>str</i> в вызывающий выходной поток
<code>void write(String str, int offset, int numChars)</code>	Записывает часть строки <i>str</i> в количестве <i>numChars</i> символов, начиная с позиции, определяемой параметром <i>offset</i> , в вызывающий поток

## Консольный ввод с использованием символьных потоков

В случае программ, подлежащих интернационализации, для ввода символов с клавиатуры проще и удобнее использовать символьные потоки, а не байтовые. Но поскольку `System.in` — это байтовый поток, для него придется построить оболочку в виде класса, производного от класса `Reader`. Наиболее подходящим для ввода с консоли является класс `BufferedReader`, поддерживающий буферизованный входной поток. Однако объект типа `BufferedReader` нельзя создать непосредственно на основе стандартного потока ввода `System.in`. Сначала нужно преобразовать байтовый поток в символьный. Для этого используется класс `InputStreamReader`, преобразующий байты в символы. Чтобы получить объект типа `InputStreamReader`, связанный с потоком стандартного ввода `System.in`, необходимо воспользоваться следующим конструктором:

```
InputStreamReader(InputStream inputStream)
```

Поток ввода `System.in` — это экземпляр класса `InputStream`, и поэтому его можно указать в качестве параметра `inputStream` данного конструктора.

Затем с помощью объекта, созданного на основе объекта типа `InputStreamReader`, можно получить объект типа `BufferedReader`, используя следующий конструктор:

```
BufferedReader(Reader inputReader)
```

где `inputReader` — поток, который связывается с создаваемым экземпляром класса `BufferedReader`. Объединяя обращения к указанным выше конструкторам в одну операцию, мы получаем приведенную ниже строку кода. В ней создается объект типа `BufferedReader`, связанный с клавиатурой.

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

После выполнения этого кода переменная `br` будет содержать ссылку на символьный поток, связанный с консолью через поток ввода `System.in`.

### Чтение символов

Чтение символов из потока `System.in` с помощью метода `read()` осуществляется в основном так, как если бы это делалось с помощью байтовых потоков. Ниже приведены общие формы объявления трех версий метода `read()`, рассмотренных в классе `BufferedReader`.

```
int read() throws IOException
int read(char data[]) throws IOException
int read(char data[], int start, int max) throws IOException
```

Первая версия метода `read()` читает одиночный символ в кодировке `Unicode`. По достижении конца потока метод возвращает значение `-1`. Вторая версия метода `read()` читает символы из входного потока и помещает их в массив. Этот процесс продолжается до тех пор, пока не будет достигнут конец потока, или пока массив `data` не заполнится символами, или не возникнет ошибка, в зависимости от того, какое из этих событий произойдет первым.

В этом случае метод возвращает число прочитанных символов, а в случае достижения конца потока — значение `-1`. Третья версия метода `read()` помещает прочитанные символы в массив `data`, начиная с элемента, определяемого параметром `start`. Максимальное число символов, которые могут быть записаны в массив, определяется параметром `max`. В данном случае метод возвращает число прочитанных символов или значение `-1`, если достигнут конец потока. При возникновении ошибки в каждой из вышеперечисленных версий метода `read()` генерируется исключение `IOException`. При чтении данных из потока ввода `System.in` конец потока устанавливается нажатием клавиши `<Enter>`.

Ниже приведен пример программы, демонстрирующий применение метода `read()` для чтения символов с консоли. Символы читаются до тех пор, пока пользователь не введет точку. Следует иметь в виду, что исключения, которые могут быть сгенерированы при выполнении данной программы, обрабатываются за пределами метода `main()`. Как уже отмечалось, такой подход к обработке ошибок является типичным при чтении данных с консоли. По желанию можно использовать другой механизм обработки ошибок.

```
// Использование класса BufferedReader
// для чтения символов с консоли
import java.io.*;

class ReadChars {
    public static void main(String args[]) throws IOException
    {
        char c;
        Создание класса BufferedReader,
        связанного с потоком System.In
        BufferedReader br = new ←—————|
            BufferedReader(new InputStreamReader(System.in));

        System.out.println("Введите символы; окончание ввода -
            символ точки");

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

Результат выполнения данной программы выглядит следующим образом.

```
Введите символы; окончание ввода - символ точки
One Two.
O
n
e
T
w
o
.
```



## Чтение строк

Для ввода строки с клавиатуры используют метод `readLine()` класса `BufferedReader`. Вот общая форма объявления этого метода:

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

Этот метод возвращает объект типа `String`, содержащий прочитанные символы. При попытке прочитать строку по достижении конца потока метод возвращает значение `null`.

Ниже приведен пример программы, демонстрирующий использование класса `BufferedReader` и метода `readLine()`. В этой программе текстовые строки читаются и отображаются до тех пор, пока не будет введено слово "stop".

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

class ReadLines {
    public static void main(String args[]) throws IOException
    {
        // Создать объект типа BufferedReader,
        // связанный с потоком System.in
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));

        String str;

        System.out.println("Введите текстовые строки");
        System.out.println("Признак конца ввода - строка 'stop' ");
        do {
            str = br.readLine(); ← Использование метода readLine() из класса
            System.out.println(str); ← BufferedRead для чтения строки текста
        } while(!str.equals("stop"));
    }
}
```

## Вывод на консоль с использованием символьных потоков

Несмотря на то что поток стандартного вывода `System.out` вполне пригоден для вывода на консоль, в большинстве случаев такой подход рекомендуется использовать лишь в целях отладки или при создании очень простых программ наподобие тех, которые приведены в данной книге в качестве примеров. В реальных прикладных программах на Java вывод на консоль обычно организуется через поток `PrintWriter`. Класс `PrintWriter` является одним из классов, представляющих символьные потоки. Как уже упоминалось, применение потоков упрощает локализацию прикладных программ.

В классе `PrintWriter` определен целый ряд конструкторов. Далее будет использоваться следующий конструктор:

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

Здесь в качестве первого параметра, *outputStream*, конструктору передается объект типа *OutputStream*, а второй параметр, *flushOnNewline*, указывает, должен ли буфер выходного потока сбрасываться каждый раз, когда вызывается (среди прочих других) метод *println()*. Если параметр *flushOnNewline* имеет значение *true*, сбрасывание буфера выполняется автоматически.

В классе *PrintWriter* поддерживаются методы *print()* и *println()* для всех типов, включая *Object*. Следовательно, методы *print()* и *println()* можно использовать точно так же, как и совместно с потоком вывода *System.out*. Если значение аргумента не относится к простому типу, то методы класса *PrintWriter* вызывают метод *toString()* для объекта, указанного в качестве параметра, а затем выводят результат.

Для вывода данных на консоль через поток типа *PrintWriter* следует указать *System.out* в качестве выходного потока и обеспечить вывод данных из буфера после каждого вызова метода *println()*. Например, при выполнении следующей строки кода создается объект типа *PrintWriter*, связанный с консолью:

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

Ниже приведен пример программы, демонстрирующий использование класса *PrintWriter* для организации вывода на консоль.

```
// Использование класса PrintWriter
import java.io.*;

public class PrintWriterDemo {
    public static void main(String args[]) {
        PrintWriter pw = new PrintWriter(System.out, true);
        int i = 10;
        double d = 123.65;

        pw.println("Использование класса PrintWriter");
        pw.println(i);
        pw.println(d);

        pw.println(i + " + " + d + " = " + (i+d));
    }
}
```

Создание класса *PrintWriter*,  
связанного с потоком *System.out*

Выполнение этой программы дает следующий результат.

```
Использование класса PrintWriter
10
123.65
10 + 123.65 = 133.65
```

Как ни удобны символьные потоки, не следует забывать, что для изучения языка Java или отладки программ вам будет вполне достаточно использовать поток *System.out*. Если вы используете поток *PrintWriter*, программу будет проще интернационализировать. Для небольших программ наподобие тех,

которые представлены в данной книге в виде примеров, использование потока `PrintWriter` не дает никаких существенных преимуществ по сравнению с потоком `System.out`, поэтому далее для вывода на консоль будет использоваться поток `System.out`.

## Файловый ввод-вывод с использованием символьных потоков

Несмотря на то что файловые операции ввода-вывода чаще всего выполняются с помощью байтовых потоков, для этой цели можно использовать также символьные потоки. Преимущество символьных потоков заключается в том, что они оперируют непосредственно символами в кодировке Unicode. Так, если вам нужно сохранить текст в кодировке Unicode, то для этой цели лучше всего воспользоваться символьными потоками. Как правило, для файлового ввода-вывода символов используются классы `FileReader` и `FileWriter`.

### Класс `FileWriter`

Класс `FileWriter` создает объект типа `Writer`, который можно использовать для записи данных в файл. Ниже приведены общие формы объявления двух наиболее часто используемых конструкторов данного класса.

```
FileWriter(String имя_файла) throws IOException
FileWriter(String имя_файла, boolean append) throws IOException
```

Здесь *имя\_файла* обозначает полный путь к файлу. Если параметр *append* имеет значение `true`, данные записываются в конец файла, в противном случае запись осуществляется поверх существующих данных. При возникновении ошибки в каждом из указанных конструкторов генерируется исключение `IOException`. Класс `FileWriter` является производным от классов `OutputStreamWriter` и `Writer`. Следовательно, в нем доступны методы, объявленные в его суперклассах.

Ниже приведен пример небольшой программы, демонстрирующий ввод текстовых строк с клавиатуры и последующую их запись в файл `test.txt`. Набираемый текст читается до тех пор, пока пользователь не введет слово "stop". Для вывода текстовых строк в файл используется класс `FileWriter`.

```
// Пример простой утилиты для ввода данных с клавиатуры и
// записи их на диск, демонстрирующий использование класса
// FileWriter

import java.io.*;

class KtoD {
    public static void main(String args[])
    {
```

```

String str;
BufferedReader br =
    new BufferedReader(new InputStreamReader(System.in));
System.out.println("Признак конца ввода - строка 'stop' ");

try (FileWriter fw = new FileWriter("test.txt")) ← Создание класса
{                                             FileWriter
    do {
        System.out.print(": ");
        str = br.readLine();

        if(str.compareTo("stop") == 0) break;

        str = str + "\r\n"; // добавить символы
                            // перевода строки
        fw.write(str); ← Запись строк в файл
    } while(str.compareTo("stop") != 0);
} catch(IOException exc) {
    System.out.println("Ошибка ввода-вывода:" + exc);
}
}

```

## Класс FileReader

Класс `FileReader` создает объект типа `Reader`, который можно использовать для чтения содержимого файла. Чаще всего используется следующая форма конструктора этого класса:

```
FileReader(String имя_файла) throws FileNotFoundException
```

где *имя\_файла* обозначает полный путь к файлу. Если указанного файла не существует, генерируется исключение `FileNotFoundException`. Класс `FileReader` является производным от классов `InputStreamReader` и `Reader`. Следовательно, в нем доступны методы, объявленные в его суперклассах.

В приведенном ниже примере создается простая утилита, отображающая на экране содержимое текстового файла `test.txt`. Она является своего рода дополнением к утилите, рассмотренной в предыдущем разделе.

```
// Пример простой утилиты для чтения данных с диска и вывода их
// на экран, демонстрирующий использование класса FileReader
```

```
import java.io.*;

class DtoS {
    public static void main(String args[]) {
        String s;

        // Создать и использовать объект FileReader, помещенный
        // в оболочку на основе класса BufferedReader
        try (BufferedReader br =
            new BufferedReader(new FileReader("test.txt"))) ←

```

```

    {
        while((s = br.readLine()) != null) {
            System.out.println(s);
        }
    } catch(IOException exc) {
        System.out.println("Ошибка ввода-вывода: " + exc);
    }
}
}
}

```

Обратите внимание на то, что для потока `FileReader` создается оболочка на основе класса `BufferedReader`. Благодаря этому появляется возможность обращаться к методу `readLine()`. Кроме того, закрытие потока типа `BufferedReader`, на который в данном примере ссылается переменная `br`, автоматически приводит к закрытию файла.

## СПРОСИМ У ЭКСПЕРТА

**ВОПРОС.** Я слышал об отдельном пакете ввода-вывода, называемом NIO, который включает классы для поддержки ввода-вывода. Что он собой представляет?

**ОТВЕТ.** Пакет NIO (сокращение от *New I/O*) был реализован в версии JDK 1.4. В нем поддерживается канальный подход к организации операций ввода-вывода. Классы новой подсистемы ввода-вывода относятся к пакету `java.nio` и его подчиненным пакетам, включая `java.nio.channels` и `java.nio.charset`.

Пакет NIO опирается на два основных понятия: *буфер* и *канал*. Буфер содержит данные, а канал представляет собой соединение, устанавливаемое с устройством ввода-вывода, например с файлом на диске или сетевым сокетом. Как правило, для применения новой подсистемы ввода-вывода нужно получить канал доступа к устройству и буфер для хранения данных. После этого можно выполнять необходимые операции с буфером, в частности, вводить или выводить данные.

Кроме буфера и канала, в NIO используются также понятия набора символов и селектора. *Набор символов* определяет способ преобразования байтов в символы. Для представления последовательности символов в виде набора байтов используется *шифратор*. Обратное преобразование осуществляет *дешифратор*. А *селектор* поддерживает неблокирующий мультиплексный ввод-вывод с ключом шифрования. Иными словами, селекторы позволяют выполнять обмен данными по нескольким каналам. Селекторы находят наибольшее применение в каналах, поддерживаемых сетевыми сокетами.

В версии JDK 7 новая подсистема ввода-вывода была значительно усовершенствована, и поэтому нередко ее называют *NIO.2*. К числу ее усовершенствований относятся три новых пакета (`java.nio.file`, `java.nio.file`).

`attribute` и `java.nio.file.spi`), ряд новых классов, интерфейсов и методов, а также непосредственная поддержка потокового ввода-вывода. Все эти дополнения в значительной степени расширили область применения NIO, и особенно это касается обработки файлов.

Однако новая подсистема ввода-вывода не призвана заменить классы ввода-вывода, существующие в пакете `java.io`. Классы NIO лишь дополняют стандартную подсистему ввода-вывода, предлагая альтернативный подход, вполне уместный в ряде случаев.

## Использование классов-оболочек для преобразования числовых строк

Прежде чем завершить обсуждение средств ввода-вывода, необходимо рассмотреть еще один прием, который оказывается весьма полезным при чтении числовых строк. Как вам уже известно, метод `println()` предоставляет удобные средства для вывода на консоль различных типов данных, включая целые числа и числа с плавающей точкой. Он автоматически преобразует числовые значения в удобную для чтения форму. Но в Java отсутствует метод, который читал бы числовые строки и преобразовывал их во внутреннюю двоичную форму. Например, не существует варианта метода `read()`, который читал бы числовую строку "100" и автоматически преобразовывал ее в целое число, пригодное для хранения в переменной типа `int`. Но для этой цели в Java имеются другие средства. И проще всего подобное преобразование осуществляется с помощью так называемых *оболочек типов* (объектных оболочек) Java.

Объектные оболочки в Java представляют собой классы, которые инкапсулируют простые типы. Оболочки типов необходимы, поскольку простые типы не являются объектами, что ограничивает их применение. Так, простой тип нельзя передать методу по ссылке. Для того чтобы исключить ненужные ограничения, в Java были предусмотрены классы, соответствующие каждому из простых типов.

Объектными оболочками являются классы `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `Character` и `Boolean`, которые предоставляют обширный ряд методов, позволяющих полностью интегрировать простые типы в иерархию объектов Java. Кроме того, в классах-оболочках числовых типов содержатся методы, предназначенные для преобразования числовых строк в соответствующие двоичные эквиваленты. Эти методы приведены ниже. Каждый из них возвращает двоичное значение, соответствующее числовой строке.

Оболочка типа	Метод преобразования
Double	static double parseDouble(String str) throws NumberFormatException
Float	static float parseFloat(String str) throws NumberFormatException
Long	static long parseLong(String str) throws NumberFormatException
Integer	static int parseInt(String str) throws NumberFormatException
Short	static short parseShort(String str) throws NumberFormatException
Byte	static byte parseByte(String str) throws NumberFormatException

Оболочки целочисленных типов также предоставляют дополнительный метод синтаксического анализа, позволяющий задавать основание системы счисления.

Методы синтаксического анализа позволяют без труда преобразовать во внутренний формат числовые значения, введенные в виде символьных строк с клавиатуры или из текстового файла. Ниже приведен пример программы, демонстрирующий применение для этих целей методов `parseInt()` и `parseDouble()`. Эта программа вычисляет среднее арифметическое ряда чисел, введенных пользователем с клавиатуры. Сначала пользователю предлагается указать количество подлежащих обработке числовых значений, а затем программа вводит числа с клавиатуры, используя метод `readLine()`, и с помощью метода `parseInt()` преобразует символьную строку в целочисленное значение. Далее осуществляется ввод числовых значений и последующее их преобразование в тип `double` с помощью метода `parseDouble()`.

```
/* Данная программа находит среднее арифметическое для
   ряда чисел, введенных пользователем с клавиатуры. */

import java.io.*;

class AvgNums {
    public static void main(String args[]) throws IOException
    {
        // Создание объекта типа BufferedReader,
        // использующего поток ввода System.in
        BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
        String str;
        int n;
        double sum = 0.0;
        double avg, t;

        System.out.print("Сколько чисел нужно ввести: ");
        str = br.readLine();
```

```

try {
    n = Integer.parseInt(str); ← Преобразование строки в тип int
}
catch(NumberFormatException exc) {
    System.out.println("Неверный формат");
    n = 0;
}

System.out.println("Ввод " + n + " значений");
for(int i=0; i < n ; i++) {
    System.out.print(": ");
    str = br.readLine();
    try {
        t = Double.parseDouble(str); ← Преобразование строки
    } catch(NumberFormatException exc) { ← в тип double
        System.out.println("Неверный формат");
        t = 0.0;
    }
    sum += t;
}
avg = sum / n;
System.out.println("Среднее значение: " + avg);
}
}

```

Выполнение этой программы может дать, например, следующий результат.

```

Сколько чисел нужно ввести: 5
Ввод 5 значений
: 1.1
: 2.2
: 3.3
: 4.4
: 5.5
Среднее значение: 3.3

```

## СПРОСИМ У ЭКСПЕРТА

**ВОПРОС.** Могут ли объектные оболочки простых типов выполнять другие функции, кроме описанных в этом разделе?

**ОТВЕТ.** Классы оболочек простых типов предоставляют ряд методов, помогающих интегрировать эти типы данных в иерархию объектов. Например, различные механизмы хранения данных, предусмотренные в библиотеке Java, включая отображения, списки и множества, взаимодействуют только с объектами. Поэтому для сохранения целочисленного значения в списке его следует сначала преобразовать в объект. Оболочки типов предоставляют также метод `compareTo()` для сравнения текущего значения с заданным, метод `equals()` для проверки равенства двух объектов, а также методы, возвращающие значение объекта в разных формах записи. К оболочкам типов мы еще вернемся в главе 12, когда речь пойдет об автоупаковке.



**Упражнение 10.2****Создание справочной системы,  
находящейся на диске**`FileHelp.java`

В упражнении 4.1 был создан класс `Help`, позволяющий отображать сведения об инструкциях Java. Справочная информация хранилась в самом классе, а пользователь выбирал требуемые сведения из меню. И хотя такая справочная система выполняет свои функции, подход к ее разработке был выбран далеко не самый лучший. Например, если потребуются добавить или изменить какие-либо сведения, вам придется внести изменения в исходный код программы, которая реализует справочную систему. Кроме того, выбирать пункт меню по его номеру не очень удобно, а если количество пунктов велико, то такой способ вообще непригоден. В этом упражнении нам предстоит устранить недостатки, имеющиеся в справочной системе, расположив справочную информацию на диске.

В новом варианте справочная информация должна храниться в файле. Это будет обычный текстовый файл, который можно изменять, не затрагивая исходный код программы. Для того чтобы получить справку по конкретному вопросу, следует ввести название темы. Система будет искать соответствующий раздел в файле. Если поиск завершится успешно, справочная информация будет выведена на экран. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте файл, в котором хранится справочная информация и который будет использоваться в справочной системе. Это должен быть обычный текстовый файл, организованный, как показано ниже.

```
#название_темы_1
Информация по теме

#название_темы_2
Информация по теме

...

#название_темы_N
Информация по теме
```

Название каждой темы располагается в отдельной строке и предваряется символом `#`. Наличие специального символа в строке (в данном случае — `#`) позволяет программе быстро найти начало раздела. Под названием темы может располагаться любая справочная информация. После окончания одного раздела и перед началом другого должна быть введена пустая строка. Кроме того, в конце строк не должно быть лишних пробелов.

Ниже приведен пример простого файла со справочной информацией, который можно использовать вместе с новой версией справочной системы. В нем хранятся сведения об инструкциях Java.

```

#if
if(условие) инструкция;
else инструкция;

#switch
switch(выражение) {
    case константа:
        последовательность инструкций
        break;
    // ...
}

#for
for(инициализация; условие; итерация) инструкция;

#while
while(условие) инструкция;

#do
do {
    инструкция;
} while (условие);

#break
break; или break метка;

#continue
continue; или continue метка;

```

Присвойте этому файлу имя `helpfile.txt`.

2. Создайте файл `FileHelp.java`.
3. Начните создание новой версии класса `Help` со следующих строк кода.

```

class Help {
    String helpfile; // имя файла справки

    Help(String fname) {
        helpfile = fname;
    }
}

```

Имя файла со справочной информацией передается конструктору класса `Help` и запоминается в переменной экземпляра `helpfile`. А поскольку каждый экземпляр класса `Help` содержит отдельную копию переменной `helpfile`, то каждый из них может взаимодействовать с отдельным файлом. Это дает возможность создавать отдельные наборы справочных файлов на разные темы.

4. Добавьте в класс `Help` метод `helpon()`, код которого приведен ниже. Этот метод извлекает справочную информацию по заданной теме.

```

// Отображение справочной информации по указанной теме
boolean helpon(String what) {

```

```

int ch;
String topic, info;

// Открыть справочный файл
try (BufferedReader helpRdr =
    new BufferedReader(new FileReader(helpfile)))
{
    do {
        // Читать символы до тех пор, пока не встретится символ #
        ch = helpRdr.read();

        // Проверить, совпадают ли темы
        if(ch == '#') {
            topic = helpRdr.readLine();
            if(what.compareTo(topic) == 0) { // найти тему
                do {
                    info = helpRdr.readLine();
                    if(info != null) System.out.println(info);
                } while((info != null) && (info.compareTo("") != 0));
                return true;
            }
        }
    } while(ch != -1);
}
catch(IOException exc) {
    System.out.println("Ошибка при попытке доступа к
        файлу справки");
    return false;
}
return false; // тема не найдена
}

```

Прежде всего обратите внимание на то, что в методе `helpon()` обрабатываются все исключения, связанные с вводом-выводом, поэтому в заголовке метода не указано ключевое слово `throws`. Благодаря такому подходу упрощается разработка методов, в которых используется метод `helpon()`. В вызывающем методе достаточно обратиться к методу `helpon()`, не заключая его вызов в блок `try/catch`.

Для открытия файла со справочной информацией предназначен класс `FileReader`, оболочкой которого является класс `BufferedReader`. В справочном файле содержится текст, и поэтому справочную систему удобнее локализовать через символьные потоки ввода-вывода.

Метод `helpon()` действует следующим образом. Символьная строка, содержащая название темы, передается методу в качестве параметра. Сначала метод открывает файл со справочной информацией. Затем в файле осуществляется поиск, т.е. проверяется совпадение содержимого переменной `what` и названия темы. Напомним, что в файле заголовок темы предваряется символом `#`, поэтому метод сначала ищет данный символ. Если символ

найден, следующее за ним название темы сравнивается с содержимым переменной `what`. Если сравниваемые строки совпадают, то отображается справочная информация по данной теме. И если заголовок темы найден, то метод `helpon()` возвращает логическое значение `true`, в противном случае — логическое значение `false`.

5. В классе `Help` содержится также метод `getSelection()`, который предлагает указать тему и возвращает строку, введенную пользователем.

```
// Получение темы справки
String getSelection() {
    String topic = "";

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

    System.out.print("Укажите тему: ");
    try {
        topic = br.readLine();
    }
    catch(IOException exc) {
        System.out.println("Ошибка при чтении с консоли");
    }
    return topic;
}
```

В теле этого метода сначала создается объект типа `BufferedReader`, который связывается с потоком вывода `System.in`. Затем в нем запрашивается название темы, которое принимается и далее возвращается вызывающей части программы.

6. Ниже приведен весь исходный код программы, реализующей справочную систему на диске.

```
/*
    Упражнение 10.2

    Справочная система, использующая дисковый файл
    для хранения информации
*/

import java.io.*;

/* В классе Help открывается файл со справочной информацией,
    выполняется поиск указанной темы, а затем отображается
    справочная информация. Обратите внимание на то, что данный
    класс обрабатывает все исключения, освобождая от этого
    вызывающий код. */
class Help {
    String helpfile; // имя справочного файла

    Help(String fname) {
        helpfile = fname;
    }
}
```

```
// Отображение справочной информации по указанной теме
boolean helpon(String what) {
    int ch;
    String topic, info;

    // Открыть справочный файл
    try (BufferedReader helpRdr =
        new BufferedReader(new FileReader(helpfile)))
    {
        do {
            // Читать символы до тех пор,
            // пока не встретится символ #
            ch = helpRdr.read();

            // Проверить, совпадают ли темы
            if(ch == '#') {
                topic = helpRdr.readLine();
                if(what.compareTo(topic) == 0) { // найти тему
                    do {
                        info = helpRdr.readLine();
                        if(info != null) System.out.println(info);
                    } while((info != null) &&
                        (info.compareTo("") != 0));
                    return true;
                }
            }
        } while(ch != -1);
    }
    catch(IOException exc) {
        System.out.println("Ошибка при попытке доступа
            к файлу справки");
        return false;
    }
    return false; // тема не найдена
}

// Получение темы справки
String getSelection() {
    String topic = "";

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

    System.out.print("Укажите тему: ");
    try {
        topic = br.readLine();
    }
    catch(IOException exc) {
        System.out.println("Ошибка при чтении с консоли");
    }
    return topic;
}
}
```

```
// Демонстрация работы справочной системы на основе файла
class FileHelp {
    public static void main(String args[]) {
        Help hlpobj = new Help("helpfile.txt");
        String topic;

        System.out.println("Воспользуйтесь справочной системой.\n" +
            "Для выхода из системы введите 'stop'.");

        do {
            topic = hlpobj.getSelection();

            if(!hlpobj.helpon(topic))
                System.out.println("Тема не найдена.\n");
        } while(topic.compareTo("stop") != 0);
    }
}
```

## СПРОСИМ У ЭКСПЕРТА

**ВОПРОС.** Имеется ли, помимо методов синтаксического анализа, определяемых в оболочках простых типов, другой простой способ преобразования числовой строки, вводимой с клавиатуры, в эквивалентную ей двоичную форму?

**ОТВЕТ.** Да, имеется. Другой способ преобразования числовой строки в ее внутреннее представление в двоичной форме состоит в использовании одного из методов, определенных в классе `Scanner` из пакета `java.util`. Этот класс читает данные, вводимые в удобном для чтения виде, преобразуя их в двоичную форму. Средствами класса `Scanner` можно организовать чтение данных, вводимых из самых разных источников, в том числе с консоли и из файлов. Следовательно, его можно использовать для чтения числовой строки, введенной с клавиатуры, присваивая полученное значение переменной. И хотя в классе `Scanner` содержится слишком много средств, чтобы описать их подробно, ниже приведены основные примеры его применения.

Для организации ввода с клавиатуры средствами класса `Scanner` необходимо сначала создать объект этого класса, связанный с потоком ввода с консоли. Для этой цели служит следующий конструктор:

```
Scanner(InputStream from)
```

Этот конструктор создает объект типа `Scanner`, который использует поток ввода, определяемый параметром *from*, в качестве источника ввода данных. С помощью этого конструктора можно создать объект типа `Scanner`, связанный с потоком ввода с консоли, как показано ниже:

```
Scanner conin = new Scanner(System.in);
```

Это оказывается возможным благодаря тому, что поток `System.in` является объектом типа `InputStream`. После выполнения этой строки кода перемен-

ную `conin` ссылки на объект типа `Scanner` можно использовать для чтения данных, вводимых с клавиатуры.

Как только будет создан объект типа `Scanner`, им нетрудно воспользоваться для чтения числовой строки, вводимой с клавиатуры. Ниже приведен общий порядок выполняемых для этого действий.

1. Определить, имеются ли вводимые данные конкретного типа, вызвав один из методов `hasNextX` класса `Scanner`, где  $X$  — нужный тип вводимых данных.
2. Если вводимые данные имеются, прочитать их, вызвав один из методов `nextX` класса `Scanner`.

Как следует из приведенного выше порядка действий, в классе `Scanner` определены две группы методов, предназначенных для чтения вводимых данных. К первой из них относятся методы `hasNextX`, в том числе `hasNextInt()` и `hasNextDouble()`. Каждый из методов `hasNextX` возвращает логическое значение `true`, если очередной элемент данных, имеющийся в потоке ввода, относится к нужному типу данных, а иначе — логическое значение `false`. Так, логическое значение `true` возвращается при вызове метода `hasNextInt()` лишь в том случае, если очередной элемент данных в потоке ввода является целочисленным значением, представленным в удобном для чтения виде. Если данные нужного типа имеются в потоке ввода, их можно прочитать, вызвав один из методов класса `Scanner`, относящихся к группе `next`, например метод `nextInt()` или `nextDouble()`. Эти методы преобразуют данные соответствующего типа из удобной для чтения формы во внутреннее их представление в двоичном виде, возвращая полученный результат. Так, для чтения целочисленного значения, введенного с клавиатуры, следует вызвать метод `nextInt()`.

В приведенном ниже фрагменте кода показано, каким образом организуется чтение целочисленного значения с клавиатуры.

```
Scanner conin = new Scanner(System.in);
int i;

if (conin.hasNextInt()) i = conin.nextInt();
```

Если ввести с клавиатуры `123`, то в результате выполнения приведенного выше фрагмента кода переменная `i` будет содержать целочисленное значение `123`.

Формально методы из группы `next` можно вызывать без предварительного вызова методов из группы `hasNext`, но делать этого все же не рекомендуется. Ведь если методу из группы `next` не удастся обнаружить данные искомого типа, то он сгенерирует исключение `InputMismatchException`. Поэтому лучше сначала убедиться, что данные нужного типа имеются в потоке ввода, вызвав подходящий метод `hasNext`, и только после этого вызывать соответствующий метод `next`.



## Вопросы и упражнения для самопроверки

1. Для чего в Java определены как байтовые, так и символьные потоки?
2. Как известно, консольные операции ввода-вывода осуществляются в текстовом виде. Почему же в Java для этой цели используются байтовые потоки?
3. Как открыть файл для чтения байтов?
4. Как открыть файл для чтения символов?
5. Как открыть файл для выполнения операций ввода-вывода с произвольным доступом?
6. Как преобразовать числовую строку "123.23" в ее двоичный эквивалент?
7. Напишите программу для копирования текстовых файлов. Видоизмените ее таким образом, чтобы все пробелы заменялись дефисами. Используйте при написании программы классы, представляющие байтовые потоки, а также традиционный способ закрытия файла явным вызовом метода `close()`.
8. Перепишите программу, созданную в предыдущем пункте, таким образом, чтобы в ней использовались классы, представляющие символьные потоки. На этот раз воспользуйтесь инструкцией `try` с ресурсами для автоматического закрытия файла.
9. К какому типу относится поток `System.in`?
10. Какое значение возвращает метод `read()` класса `InputStream` по достижении конца потока?
11. Поток какого типа используется для чтения двоичных данных?
12. Классы `Reader` и `Writer` находятся на вершине иерархии классов \_\_\_\_\_.
13. Инструкция `try` с ресурсами служит для \_\_\_\_\_.
14. Верно ли следующее утверждение: "Если для закрытия файла применяется традиционный способ, то лучше всего делать это в блоке `finally`"?