Работа с файлами

Чтение текстовых файлов

В языке Java ввод-вывод организован с помощью потоков данных (байтов или символов). Для каждого вида потока существует свой класс и/или интерфейс.

Основными классами для чтения информации из файла являются классы **FileInputStream** и **FileReader**. Объекты данных классов выполняют базовые методы файлового ввода-вывода, такие как открытие и закрытие файла, чтение информации из файла. Отличие между этими двумя классами заключается в способе чтения информации. Класс *FileInputStream* читает из файла отдельные байты, в то время как *FileReader* читает из файла символы. Так как все символы файла могут быть записаны либо в виде одного байта, либо в виде многобайтовых символов, то в первом случае разницы между этими классами в чтении информации нет, во втором случае результат работы объектов этих классов будет разным.

Файл открывается при создании объекта. Для этого конструктору может быть передано имя файла в виде строки или объект типа File, содержащий описание требуемого файла.

```
FileInputStream fileInputStream = new FileInputStream("hello.txt");
```

Если файл не открывается, вызывается исключение **FileNotFoundException**, созданное на основе более общего IOException. Соответственно, если необходимо выполнить обработку исключения в случае, если требуемый файл не найден, следует создать обработчик этого исключения:

```
FileInputStream fileInputStream;
try {
        fileInputStream = new FileInputStream("hello.txt");
} catch (FileNotFoundException e) {
        System.out.print("File not found");
}
```

Если обработчик исключения в данном методе не создается, следует передать исключение “наверх”, указав в заголовке: *throws FileNotFoundException*, либо *throws IOException*, так как обработчик родительского исключения перехватывает все дочерние. Когда работа с файлом завершена, следует закрыть файл, используя метод **close()**.

Чтение из потока выполнятся с помощью метода **read()**. Если этот метод вызван без параметров, он возвращает значение типа *int*, содержащее величину очередного прочитанного байта или символа, либо если файл закончился, возвращает значение «-1».

Пример чтения текстового файла и вывода его на экран. Следует обратить внимание, что в данном примере используется класс *FileReader*, поэтому он будет работать корректно и с русским текстом.

```
import java.io.*;
public class IOMain {
	public static void main(String[] args)
	throws IOException {
		FileReader reader = new FileReader("hello.txt");
		int res = reader.read();
		while( res != -1) {
			System.out.print((char)res); res = reader.read();
		}
	}
}
```

Объекты классов *FileInputStream* и *FileReader* имеют некоторые недостатки. Во-первых, это исключительно побайтовое или посимвольное чтение информации, а во-вторых, это достаточно низкое быстродействие, так как для чтения каждого байта выполняется отдельная операция доступа к диску.

Чтобы повысить быстродействие, используются классы **BufferedInputStream** вместо *FileInputStream* или **BufferedReader** вместо *FileReader*. Главным их отличием является то, что необходимые данные загружаются во внутренний буфер блоками и выдаются по мере необходимости. Таким образом быстродействие значительно увеличивается.

Чтобы не реализовывать повторно файловые операции, уже созданные в *FileInputStream* и *FileReader*, данные классы сделаны в виде обертки. То есть в *BufferedInputStream* имеется поле, хранящее ссылку на объект *FileInputStream*, который непосредственно выполняет ввод-вывод. Для другой пары классов — аналогично. Поэтому при создании объекта *BufferedInputStream* его конструктору надо указать объект *FileInputStream*. Обычно создание встроенного объекта выполняется тут же:

```
BufferedInputStream bufferedInputStream = new BufferedInputStream(new
FileInputStream("hello.txt"));
```

аналогично:

```
BufferedReader bufferedReader = new BufferedReader(new

FileReader("hello.txt"));
```

Большинство методов классов *BufferedInputStream* и *BufferedReader* работают аналогично методам внутренних классов, то есть можно использовать те же методы *read*, *close* и т.д. Но у класса *BufferedReader* есть метод, который может существенно облегчить работу с текстовыми файлами: метод для чтения строки из файла **readLine()**. Данный метод возвращает объект типа String, содержащий очередную прочитанную строку файла. Если достигнут конец

файла, возвращается значение null. Таким образом, блок чтения файла из предыдущего примера может принять вид:

```
String res = bufferedReader.readLine();
while( res != null){
        System.out.println(res);
        res = bufferedReader.readLine();
}
```

Следует обратить внимание, что при чтении строки символы завершения строки обрезаются, поэтому при выводе используется println.

Чтение двоичных файлов

Если файл является двоичным и хранит набор вещественных или целочисленных значений, использование приведенных выше классов для ввода данных очень неудобно. Классы *FileReader* и *BufferReader* вообще не пригодны для данной задачи, а использование *FileInputStream* потребует дополнительных операций, чтобы собрать значения из последовательности байтов. Поэтому был создан класс **DataInputStream**, умеющий выполнять требуемые действия. Данный класс также является оберткой и может содержать объект типа *FileInputStream* или *BufferedInputStream* (обычно используют второй для повышения быстродействия). Точно также конструктор данного класса требует объект, который непосредственно будет заниматься вводом-выводом. В результате при использовании *BufferedInputStream* создание объекта будет иметь вид:

```
DataInputStream in5 = new DataInputStream( new BufferedInputStream( new FileInputStream("Data.txt")));
```

Кроме методов, полученных от предыдущих классов, он имеет следующие:

- boolean **readBoolean**() — возвращает прочитанное из файла булевское значение;
- byte **readByte**() — возвращает прочитанный из файла байт;
- char **readChar**() — возвращает символ;
- double **readDouble**() — возвращает вещественное число удвоенной точности;
- float **readFloat**() — возвращает вещественное число;
- int **readInt**() — возвращает целое число;

- String **readLine**() — возвращает текстовую строку (не выполняет корректного преобразования символов, поэтому не рекомендуется к использованию);
- long **readLong**() — возвращает длинное целое;
- short **readShort**() — возвращает короткое целое;
- int **readUnsignedByte**() – возвращает беззнаковое значение байта;
- int **readUnsignedShort**() – возвращает беззнаковое значение короткого целого;
- int **skipBytes**(int n) – пропускает в файле указанное количество байтов.

Следует обратить внимание, что у данных методов отсутствует возможность сообщить о достижении конца файла с помощью возвращаемого значения. Поэтому, если достигнут конец файла, они создают исключение *EOFException*. Это значит, что для корректной обработки файла его следует перехватывать.

Например, вывод на экран файла, содержащего набор целочисленных значений, будет выглядеть:

```
import java.io.*; public class IOMain {
    public static void main(String[] args) throws IOException {
        DataInputStream dataInputStream = null;
        try{
            dataInputStream = new DataInputStream(new
            BufferedInputStream( new FileInputStream("hello.dat")));
        } catch (FileNotFoundException e) {
            System.out.print("File not found");
            return;
        }
        int res = dataInputStream.readInt(); while(true) {
            System.out.println(res);
            try {
                res = dataInputStream.readInt();
            } catch (EOFException e) {
                break;
            }
        }
        if(dataInputStream != null ){
            dataInputStream.close();
```

```
            }
        }
}
```

Запись в файл

Процесс записи информации в файл выполняется аналогично чтению, для записи также имеются специальные классы. Базовыми классами являются **FileOutputStream** и **FileWriter**. Создание объектов этих классов происходит точно также, как и для классов чтения:

```
FileOutputStream f = new FileOutputStream("hello.dat");
```

Если файл открыть не удается, происходит создание исключения IOException. Кроме того, у конструкторов может быть второй параметр булевского типа. Если он равен истине, происходит открытие файла на добавление, то есть содержимое файла не удаляется.

Для записи в файл у данных классов используется метод **write**. Он выполняет побайтовую запись у класса *FileOutputStream* и посимвольную у класса *FileWriter*. В качестве параметра может быть передан один байт (символ) либо массив. Если передан массив, можно также указать еще два числовых параметра: начиная с какого элемента и сколько элементов массива следует записать в файл. У класса FileWriter передаваемые для записи данные могут быть в виде объекта String.

Если записать информацию невозможно, создается исключение IOException. Точно так же для увеличения быстродействия используются объекты BufferedWriter или BufferedOutputStream. Так как эти классы могут не сразу записывать информацию на диск, для того чтобы сделать это принудительно, в них имеется метод **flush()**.

Для записи в файл непосредственно значений переменных используются классы **PrintWriter** и **DataOutputStream**. Первый используется для записи данных в символьном виде, второй — в двоичном.

Как и для чтения, при создании объектов данных классов конструктору следует указать объект, который непосредственно будет заниматься выводом информации:

```
DataOutputStream a = new DataOutputStream(new BufferedOutputStream(new
FileOutputStream("hello.dat")));
```

Для текстового вывода используются методы **print** и **println** класса PrintWriter. Отличие между этими методами очевидно: второй завершает вывод переводом строки. В качестве параметра этим методам может быть передана величина практически любого примитивного типа, а также строка или массив символов.

Особенность класса *PrintWriter* в том, что методы *println* и *print* не отслеживают многие ошибки, поэтому был введен специальный метод **checkError()**, который занимается проверкой, возникали ли ошибочные состояния, и возвращает *true*, если ошибка произошла.

Для двоичного вывода используется методы класса DataOutputStream:

- void **writeBoolean**(boolean v) — записывает в файл логическое значение;
- void **writeByte**(int v) — записывает в файл байт;
- void **writeBytes**(String s) — записывает в файл строку как последовательность байтов;
- void **writeChar**(int v) — записывает в файл символ;
- void **writeChars**(String s) — записывает в файл строку как последовательность символов;
- void **writeDouble**(double v) — записывает в файл вещественное число удвоенной точности;
- void **writeFloat**(float v) — записывает вещественное число;
- void **writeInt**(int v) — записывает в файл целое число;
- void **writeLong**(long v) — записывает в файл длинное целое;
- void **writeShort**(int v) — записывает в файл короткое целое.

Пример записи в файл чисел от 0 до 19 в двоичном виде:

```
DataOutputStream a;
try {
        a = new DataOutputStream(new BufferedOutputStream(new
        FileOutputStream("hello.dat")));
} catch (FileNotFoundException e) {
        System.out.print("File not found");
        return;
}
for(int i=0; i<20;i++) {
        a.writeInt(i);
}
```

Конструкция try-with-resources

В процессе работы с файлами может возникнуть ситуация неправильного освобождения ресурсов. Если вы открываете много файлов, но не закрываете их, то они остаются в памяти, что может привести к утечке. В результате программа может перестать работать, когда закончится оперативная память.

Чтобы этого не произошло, используется конструкция try-with-resources. Так как потоки с файлами реализуют интерфейс java.io.Closeable, то эта конструкция позволяет автоматически закрывать ресурсы без необходимости ручного вмешательства.

Закрытие ресурса вручную:

```
import java.io.*; public class IOMain {
        public static void main(String[] args)
        throws IOException {
                FileReader reader;
                try{
                        reader = new FileReader("hello.txt");
                        int res = reader.read();
                        while( res != -1) {
                                System.out.print((char)res);
                                res = reader.read();
                        }
                }
                catch (FileNotFoundException e) {
                        System.out.print("File not found");
                }
                finally {
                        reader.close();
                }
        }
}
```

Закрытие ресурса с помощью конструкции try-with-resources:

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

```
            try(FileReader reader = new FileReader("hello.txt");){
                int res = reader.read();
                while( res != -1) {
                    System.out.print((char)res);
                    res = reader.read();
                }
            }
            catch (IOException e) {
                System.out.print("Something wrong with file");
            }
        }
}
```

Класс File

Для некоторых базовых операций с файлами используется класс **File**. Несмотря на свое название этот класс позволяет работать не только с файлами, но и с каталогами. При создании объекта данного класса ему в соответствие ставится имя файла или каталога. Для этого конструктору передается строка с именем. Имя может быть как абсолютным (то есть с полным указанием пути), так и относительным (то есть относительно текущего каталога). Если необходимо в File занести текущий каталог, его обозначают точкой, например:

```
File path = new File(".");
```

Так как в разных операционных системах используются различные обозначения для разделителей каталогов, в классе имеется специальное свойство pathSeparator, хранящее в виде строки знак, разделяющий имена каталогов в пути. Для работы с файлами и каталогами имеется следующие методы:

- boolean **canRead**() — проверяет возможность прочитать файл, на который указывает объект;
- boolean **canWrite**() — проверяет, может ли выполняться запись в файл, на который указывает объект;
- int **compareTo**(File pathname) — сравнивает путь к данному файлу и путь переданного ему файла как строки;
- boolean **createNewFile**() — создает новый файл, имя которого задано данным объектом. Если файл уже существует, операция не выполняется, и возвращается значение «false», если файл создан, возвращается «true»;

- static File **createTempFile**(String prefix, String suffix) — создает пустой файл в каталоге для временных файлов. При задании имени используются переданные строки префикса и суффикса. Третьим параметром может быть передан объект **File**, задающий каталог, где надо создать файл;
- boolean **delete**() — удаляет файл или каталог, заданный данным объектом;
- void **deleteOnExit**() — требует от виртуальной машины удаления этого файла, когда работа будет завершена;
- boolean **exists**() — проверяет, существует ли файл или каталог, заданный этим объектом;
- File **getAbsoluteFile**() — возвращает объект, содержащий абсолютный путь к данному файлу;
- String **getAbsolutePath**() — возвращает абсолютный путь к данному файлу в виде строки;
- File **getCanonicalFile**() — предыдущие методы хоть и возвращают абсолютный путь к файлу, но этот путь может содержать ненужные знаки, например точки или многоточия. Данный метод создает объект, хранящий путь в «каноническом» виде;
- String **getCanonicalPath**() — возвращает путь к файлу в каноническом виде как строку.

Пример:

```
File f1 = new File("./hello.txt");
System.out.println(f1.getAbsolutePath());
System.out.println(f1.getCanonicalPath());
```

Результатом работы этого фрагмента программы могут быть строки (пример для Linux):

/home/academy/install/eclipse/workspace/ioproj/./hello.txt

/home/academy/install/eclipse/workspace/ioproj/hello.txt

- String **getName**() — возвращает имя файла или каталога, содержащегося в этом объекте;
- String **getParent**() — возвращает путь к родительскому каталогу;
- File **getParentFile**() — возвращает объект, содержащий родительский каталог;
- String **getPath**() — возвращает путь к файлу в виде строки;
- boolean **isAbsolute**() — проверяет, является ли путь, задающий данный файл, абсолютным;

- boolean **isDirectory**() — проверяет, указывает ли данный объект на каталог;
- boolean **isFile**() — проверяет, указывает ли данный объект на файл;
- boolean **isHidden**() — проверяет, является ли данный файл или каталог скрытым;
- long **lastModified**() — возвращает время, когда данный файл был изменен последний раз;
- long **length**() — возвращает длину файла;
- String[] **list**() — если объект указывает на каталог, возвращает массив строк с именами файлов и каталогов, находящихся в нем;
- File[] **listFiles**() — если объект указывает на каталог, возвращает массив объектов, связанных с файлами и каталогами, находящимися в нем;
- static File[] **listRoots**() — возвращает допустимые корни файловой системы;
- boolean **mkdir**() — создает каталог, на который указывает данный объект;
- boolean **mkdirs**() — аналогично предыдущему, но также создает, если необходимо, родительские каталоги;
- boolean **renameTo**(File dest) — переименовывает файл;
- boolean **setLastModified**(long time) — устанавливает время, когда файл последний раз изменялся;
- boolean **setReadOnly**() — устанавливает для файла атрибут «только для чтения».