

JAVA. Лабораторная работа №5

Тема: Ввод-вывод текстовых файлов. Javadoc

1.1 Цель работы

Закрепить навыки работы с текстовыми потоками и файлами, с разбором строки на отдельные элементы, с методами преобразования строковых данных в числовые и числовых в строковые.

1.2 Теория – потоки ввода/вывода

Программы, написанные нами в предыдущих лабораторных работах, воспринимали информацию только из параметров командной строки и графических компонентов, а результаты выводили на консоль или в графические компоненты. Однако во многих случаях требуется выводить результаты на принтер, в файл, базу данных или передавать по сети. Исходные данные тоже часто приходится загружать из файла, базы данных или из сети.

Для того чтобы отвлечься от особенностей конкретных устройств ввода/вывода, в Java употребляется понятие потока(stream). Считается, что в программу идет входной поток(input stream) символов Unicode или просто байтов, воспринимаемый в программе методами read(). Из программы методами write() или print(), println() выводится выходной поток(output stream) символов или байтов. При этом неважно, куда направлен поток: на консоль, на принтер, в файл или в сеть, методы write() и print() ничего об этом не знают.

Можно представить себе поток как трубу, по которой в одном направлении последовательно "текут" символы или байты, один за другим. Методы read(), write(), print(), println() взаимодействуют с одним концом трубы, другой конец соединяется с источником или приемником данных конструкторами классов, в которых реализованы эти методы.

Конечно, полное игнорирование особенностей устройств ввода/вывода сильно замедляет передачу информации. Поэтому в Java все-таки выделяется файловый ввод/вывод, вывод на печать, сетевой поток.

Три потока определены в классе System статическими полями **in**, **out** и **err**. Они называются соответственно стандартным вводом(stdin), стандартным выводом(stdout) и стандартным выводом сообщений(stderr). Эти стандартные потоки могут быть соединены с разными конкретными устройствами ввода и вывода.

Потоки out и err — это экземпляры класса PrintStream, организующего выходной поток байтов. Эти экземпляры выводят информацию на консоль методами print(), println() и write(), которых в классе PrintStream имеется около двадцати для разных типов аргументов.

Поток err предназначен для вывода системных сообщений программы: трассировки, сообщений об ошибках или, просто, о выполнении каких-то этапов программы. Такие сведения обычно заносятся в специальные журналы, log-файлы, а не выводятся на консоль. В Java есть средства переназначения потока, например, с консоли в файл.

Поток in — это экземпляр класса InputStream. Он назначен на клавиатурный ввод с консоли методами read(). Класс InputStream абстрактный, поэтому реально используется какой-то из его подклассов.

Еще один вид потока — поток байтов, составляющих объект Java. Его можно

направить в файл или передать по сети, а потом восстановить в оперативной памяти. Эта операция называется **сериализацией(serialization)** объектов.

Методы организации потоков собраны в классы пакета **java.io**. Кроме классов, организующих поток, в пакет **java.io** входят классы с методами преобразования потока, например, можно преобразовать поток байтов, образующих целые числа, в поток этих чисел.

Еще одна возможность, предоставляемая классами пакета **java.io**, — слить несколько потоков в один поток.

Итак, в Java есть целых четыре иерархии классов для создания, преобразования и слияния потоков. Во главе иерархии четыре класса, непосредственно расширяющих класс **Object**:

Reader — абстрактный класс, в котором собраны самые общие методы символьного ввода;

Writer — абстрактный класс, в котором собраны самые общие методы символьного вывода;

InputStream — абстрактный класс с общими методами байтового ввода;

OutputStream — абстрактный класс с общими методами байтового вывода.

Классы входных потоков **Reader** и **InputStream** определяют по три метода ввода:

read() — возвращает один символ или байт, взятый из входного потока, в виде целого значения типа **int**; если поток уже закончился, возвращает **-1**;

read(char[] buf) — заполняет заранее определенный массив **buf** символами из входного потока; в классе **InputStream** массив типа **byte[]** и заполняется он байтами; метод возвращает фактическое число взятых из потока элементов или **-1**, если поток уже закончился;

read(char[] buf, int offset, int len) — заполняет часть символьного или байтового массива **buf**, начиная с индекса **offset**, число взятых из потока элементов равно **len**; метод возвращает фактическое число взятых из потока элементов или **-1**.

Эти методы выбрасывают **IOException**, если произошла ошибка ввода/вывода.

Четвертый метод **skip(long n)** "проматывает" поток с текущей позиции на **n** символов или байтов вперед. Эти элементы потока не вводятся методами **read()**. Метод возвращает реальное число пропущенных элементов, которое может отличаться от **n**, например поток может закончиться.

Классы выходных потоков **Writer** и **OutputStream** определяют по три почти одинаковых метода вывода:

write(char[] buf) — выводит массив в выходной поток, в классе **OutputStream** массив имеет тип **byte[]**;

write(char[] buf, int offset, int len) — выводит **len** элементов массива **buf**, начиная с элемента с индексом **offset**;

write(int elem) в классе **Writer** - выводит 16, а в классе **OutputStream** 8 младших битов аргумента **elem** в выходной поток,

В классе **Writer** есть еще два метода:

write(String s) — выводит строку **s** в выходной поток;

write(String s, int offset, int len) — выводит **len** символов строки **s**, начиная с символа с номером **offset**.

Наконец, по окончании работы с потоком его необходимо закрыть методом **close()**.

Классы, входящие в иерархии потоков ввода/вывода, показаны на рис. 1 и 2.

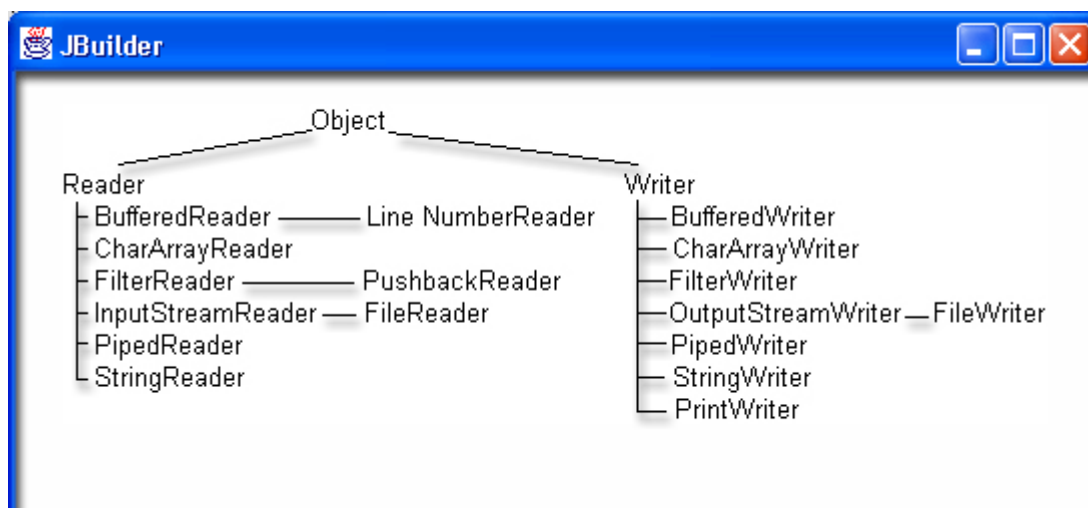


Рис. 1 – Иерархия символьных потоков

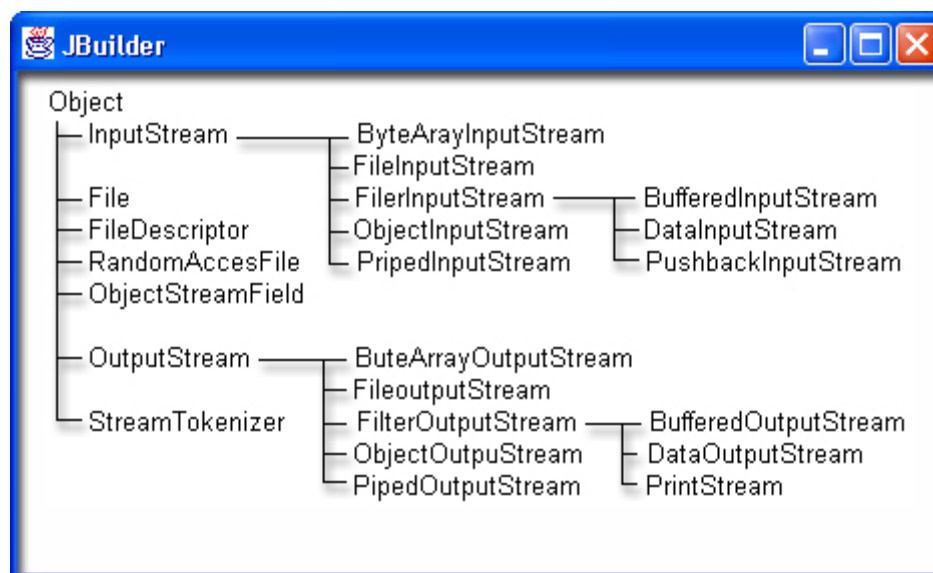


Рис. 2 – Классы байтовых потоков

Все классы пакета java.io можно разделить на две группы: классы, **создающие поток (data sink)**, и классы, **управляющие потоком (data processing)**.

Классы, создающие потоки, в свою очередь, можно разделить на пять групп:

- классы, создающие потоки, связанные с файлами (FileReader FileInputStream FileWriterFile OutputStream RandomAccessFile);
- классы, создающие потоки, связанные с массивами: (CharArrayReader ByteArrayInputStream CharArrayWriter ByteArrayOutputStream);
- классы, создающие каналы обмена информацией между подпроцессами: (PipedReader PipedInputStream PipedWriter PipedOutputStream);
- классы, создающие символьные потоки, связанные со строкой: (StringReader StringWriter);
- классы, создающие байтовые потоки из объектов Java: (ObjectInputStream ObjectOutputStream)

Классы, управляющие потоком, получают в своих конструкторах уже имеющийся поток и создают новый, преобразованный поток. Можно представлять их

себе как "переходное кольцо", после которого идет труба другого диаметра.

Четыре класса созданы специально для преобразования потоков: (FilterReader FilterInputStream FilterWriter FilterOutputStream).

Сами по себе эти классы бесполезны — они выполняют тождественное преобразование. Их следует расширять, переопределяя методы ввода/вывода. Но для байтовых фильтров есть полезные расширения, которым соответствуют некоторые символьные классы. Перечислим их.

Четыре класса выполняют буферизованный ввод/вывод: BufferedReader BufferedInputStream BufferedWriter BufferedOutputStream

Два класса преобразуют поток байтов, образующих восемь простых типов Java, в эти самые типы: (DataInputStream DataOutputStream)

Два класса связаны с выводом на строчные устройства — экран дисплея, принтер: (PrintWriter PrintStream)

Два класса связывают байтовый и символьный потоки:

- InputStreamReader — преобразует входной байтовый поток в символьный поток;

- OutputStreamWriter — преобразует выходной символьный поток в байтовый поток.

LineNumberReader, "умеющий" читать выходной символьный поток построчно. Строки в потоке разделяются символами '\n' и/или '\r'.

Этот обзор классов ввода/вывода немного проясняет положение, но не объясняет, как их использовать. Перейдем к рассмотрению реальных ситуаций.

Консольный ввод/вывод

Для вывода на консоль мы всегда использовали метод `println` класса `PrintStream`, никогда не определяя экземпляры этого класса. Мы просто использовали статическое поле `out` класса `System`, которое является объектом класса `PrintStream`. Исполняющая система Java связывает это поле с консолью.

Кстати говоря, если вам надоело писать `System.out.println`, то вы можете определить новую ссылку на `System.out`, например:

```
PrintStream pr = System.out;  
и писать просто pr.println().
```

Консоль является байтовым устройством, и символы Unicode перед выводом на консоль должны быть преобразованы в байты. Для символов Latin 1 с кодами '\u0000' — '\u00FF' при этом просто откидывается нулевой старший байт и выводятся байты '0x00' — '0xFF'. Для кодов кириллицы, которые лежат в диапазоне '\u0400' — '\u04FF' кодировки Unicode, и других национальных алфавитов производится преобразование по кодовой таблице, соответствующей установленной на компьютере локали.

Трудности с отображением кириллицы возникают, если вывод на консоль производится в кодировке, отличной от локали. Именно так происходит в русифицированных версиях MS Windows NT/2000. Обычно в них устанавливается локаль с кодовой страницей CP1251, а вывод на консоль происходит в кодировке CP866.

В этом случае надо заменить `PrintStream`, который не может работать с символьным потоком, на `PrintWriter` и "вставить переходное кольцо" между потоком символов Unicode и потоком байтов `System.out`, выводимых на консоль, в виде объекта класса `OutputStreamWriter`. В конструкторе этого объекта следует указать нужную кодировку, в данном случае, CP866. Все это можно сделать одним оператором:

```
PrintWriter pw = new PrintWriter(  
new OutputStreamWriter(System.out, "Cp866"), true);
```

Класс `PrintStream` буферизует выходной поток. Вторым аргументом `true` его конструктора вызывает принудительный сброс содержимого буфера в выходной поток после каждого выполнения метода `println()`. Но после `print()` буфер не сбрасывается! Для сброса буфера после каждого `print()` надо писать `flush()`, как это сделано в листинге 2.

Замечание:

Методы класса `PrintWriter` по умолчанию не очищают буфер, а метод `print()` не очищает его в любом случае. Для очистки буфера используйте метод `flush()`.

После этого можно выводить любой текст методами класса `PrintWriter`, которые просто дублируют методы класса `PrintStream`, и писать, например,

```
pw.println("Это русский текст");
```

как показано в листинге 1 и на рис. 3.

Следует заметить, что конструктор класса `PrintWriter`, в котором задан байтовый

поток, всегда неявно создает объект класса `OutputStreamWriter` с локальной кодировкой для преобразования байтового потока в символьный поток.

Ввод с консоли производится методами `read` о класса `InputStream` с помощью статического поля `in` класса `System`. С консоли идет поток байтов, полученных из scan-кодов клавиатуры. Эти байты должны быть преобразованы в символы Unicode такими же кодовыми таблицами, как и при выводе на консоль. Преобразование идет по той же схеме — для правильного ввода кириллицы удобнее всего определить экземпляр класса `BufferedReader`, используя в качестве "переходного кольца" объект класса `InputStreamReader`:

```
BufferedReader br = new BufferedReader(  
    new InputStreamReader(System.in, "Cp866"));
```

Класс `BufferedReader` переопределяет три метода `read` о своего суперкласса `Reader`. Кроме того, он содержит метод `readLine()`.

Метод `readLine` о возвращает строку типа `String`, содержащую символы входного потока, начиная с текущего, и заканчивая символом `'\n'` и/или `'\r'`. Эти символы-разделители не входят в возвращаемую строку. Если во входном потоке нет символов, то возвращается `null`.

В листинге.1 приведена программа, иллюстрирующая перечисленные методы консольного ввода/вывода. На рис. 3 показан вывод этой программы.

Листинг.1. Консольный ввод/вывод

```
import java.io.*; class PrWr{  
    public static void main(String[] args){ try{  
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in,  
"Cp866")); PrintWriter pw = new PrintWriter( new OutputStreamWriter(System.out,  
"Cp866"), true);  
        String s = "Это строка с русским текстом"; System.out.println("System.out puts: " +  
s); pw.println("PrintWriter puts: " + s);  
        int c = 0; pw.println("Посимвольный ввод:"); while((c = br.read()) != -1)  
pw.println((char)c); pw.println("Построчный ввод:"); do{  
            s = br.readLine(); pw.println(s);  
        }while(!s.equals("q"));  
        }catch(Exception e){ System.out.println(e);  
        }  
    }  
}
```

Поясним рис.3. Первая строка выводится потоком `System.out`. Как видите, кириллица выводится неправильно. Следующая строка предварительно преобразована в поток байтов, записанных в кодировке CP866.

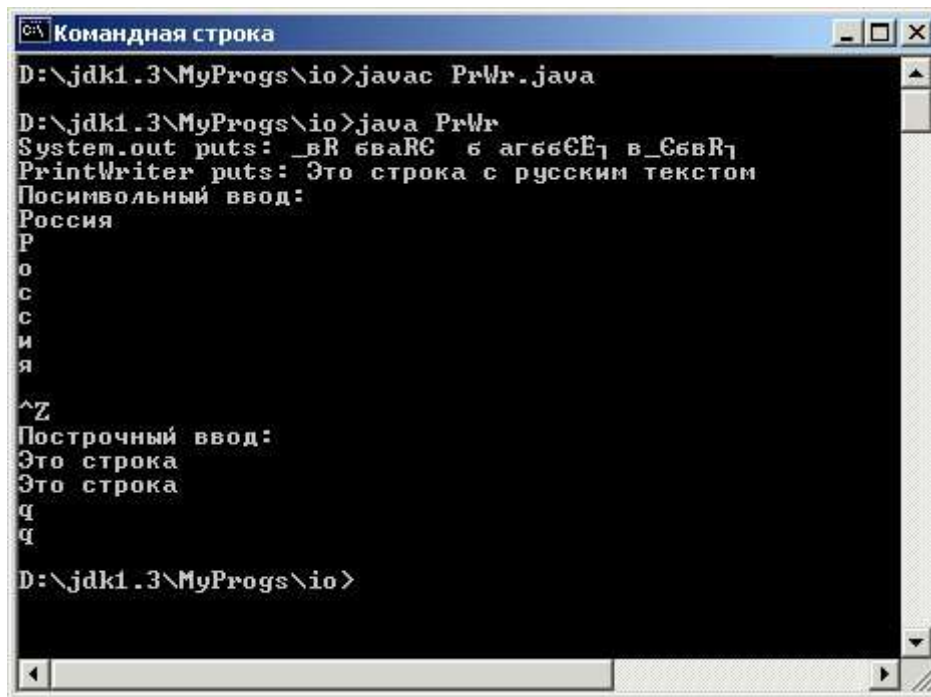
Затем, после текста "Посимвольный ввод:" с консоли вводятся символы "Россия" и нажимается клавиша `<Enter>`. Каждый вводимый символ отображается на экране — операционная система работает в режиме так называемого "эха". Фактический ввод с консоли начинается только после нажатия клавиши `<Enter>`, потому что клавиатурный ввод буферизуется операционной системой. Символы сразу после ввода отображаются

по одному на строке. Обратите внимание на две пустые строки после буквы я. Это выведены символы '\п' и '\г', которые попали во входной поток при нажатии клавиши <Enter>. У них нет никакого графического начертания(glyph).

Потом нажата комбинация клавиш <Ctrl>+<Z>. Она отображается на консоль как "^Z" и означает окончание клавиатурного ввода, завершая цикл ввода символов. Коды этих клавиш уже не попадают во входной поток.

Далее, после текста "Построчный ввод:" с клавиатуры набирается строка "Это строка" и, вслед за нажатием клавиши <Enter>, заносится в строку s. Затем строка s выводится обратно на консоль.

Для окончания работы набираем q и нажимаем клавишу <Enter>.



```
Командная строка
D:\jdk1.3\MyProgs\io>javac PrWr.java
D:\jdk1.3\MyProgs\io>java PrWr
System.out puts: _вР бваР€ б агбб€Е₁ в_ббвR₁
PrintWriter puts: Это строка с русским текстом
Посимвольный ввод:
Р
о
с
с
и
я
^Z
Построчный ввод:
Это строка
Это строка
q
q
D:\jdk1.3\MyProgs\io>
```

Рис. 3. Консольный ввод/вывод

Файловый ввод/вывод

Поскольку файлы в большинстве современных операционных систем понимаются как последовательность байтов, для файлового ввода/вывода создаются байтовые потоки с помощью классов `FileInputStream` и `FileOutputStream`. Это особенно удобно для бинарных файлов, хранящих байт-коды, архивы, изображения, звук.

Но очень много файлов содержат тексты, составленные из символов. Несмотря на то, что символы могут храниться в кодировке `Unicode`, эти тексты чаще всего записаны в байтовых кодировках. Поэтому и для текстовых файлов можно использовать байтовые потоки. В таком случае со стороны программы придется организовать преобразование байтов в символы и обратно.

Чтобы облегчить это преобразование, в пакет `java.io` введены классы `FileReader` и `FileWriter`. Они организуют преобразование потока: со стороны программы потоки символьные, со стороны файла — байтовые. Это происходит потому, что данные классы расширяют классы `InputStreamReader` и `OutputStreamWriter`, соответственно, значит, содержат "переходное кольцо" внутри себя.

Несмотря на различие потоков, использование классов файлового ввода/вывода очень похоже.

В конструкторах всех четырех файловых потоков задается имя файла в виде строки типа `string` или ссылка на объект класса `File`. Конструкторы не только создают объект, но и ищут файл и открывают его. Например:

```
FileInputStream fis = new FileInputStream("PrWr.Java");
FileReader fr = new
FileReader("D:\\jdk1.3\\src\\PrWr.Java");
```

При неудаче выбрасывается исключение класса `FileNotFoundException`, но конструктор класса

`FileWriter` выбрасывает более общее исключение `IOException`.

После открытия выходного потока типа `FileWriter` или `FileOutputStream` содержимое файла, если он был не пуст, стирается. Для того чтобы можно было делать запись в конец файла, и в том и в другом классе предусмотрен конструктор с двумя аргументами. Если второй аргумент равен `true`, то происходит дозапись в конец файла, если `false`, то файл заполняется новой информацией. Например:

```
FileWriter fw = new FileWriter("ch18.txt", true);
FileOutputStream fos = new FileOutputStream("D:\\samples\\newfile.txt");
```

Внимание:

Содержимое файла, открытого на запись конструктором с одним аргументом, стирается. Сразу после выполнения конструктора можно читать файл:

```
fis.read(); fr.read(); //или записывать в него:
fos.write((char)c); fw.write((char)c);
```

По окончании работы с файлом поток следует закрыть методом `close()`.

Преобразование потоков в классах `FileReader` и `FileWriter` выполняется по

кодовым таблицам установленной на компьютере локали. Для правильного ввода кириллицы надо применять `FileReader`, а не `FileInputStream`. Если файл содержит текст в кодировке, отличной от локальной кодировки, то придется вставлять "переходное кольцо" вручную, как это делалось для консоли, например:

```
InputStreamReader isr = new InputStreamReader(fis, "KOI8_R");
```

Байтовый поток `fis` определен выше.

Получение свойств файла

В конструкторах классов файлового ввода/вывода, описанных в предыдущем разделе, указывалось имя файла в виде строки. При этом оставалось неизвестным, существует ли файл, разрешен ли к нему доступ, какова длина файла.

Получить такие сведения можно от предварительно созданного экземпляра класса `File`, содержащего сведения о файле. В конструкторе этого класса `File(String filename)` указывается путь к файлу или каталогу, записанный по правилам операционной системы. В UNIX имена каталогов разделяются наклонной чертой `/`, в MS Windows — обратной наклонной чертой `\`, в Apple Macintosh — двоеточием `:`. Этот символ содержится в системном свойстве `file.separator` (см. рис. 6.2). Путь к файлу предваряется префиксом. В UNIX это наклонная черта, в MS Windows — буква раздела диска, двоеточие и обратная наклонная черта. Если префикса нет, то путь считается относительным и к нему прибавляется путь к текущему каталогу, который хранится в системном свойстве `user.dir`.

Конструктор не проверяет, существует ли файл с таким именем, поэтому после создания объекта следует это проверить логическим методом `exists()`.

Класс `File` содержит около сорока методов, позволяющих узнать различные свойства файла или каталога. Прежде всего, логическими методами `isFileO`, `isDirectoryO` можно выяснить, является ли путь, указанный в конструкторе, путем к файлу или каталогу.

Для каталога можно получить его содержимое — список имен файлов и подкаталогов — методом `list` о, возвращающим массив строк `stringf[]`. Можно получить такой же список в виде массива объектов класса `File[]` методом `listFilest`). Можно выбрать из списка только некоторые файлы, реализовав интерфейс `FileNameFiiter` и обратившись к методу `list(File NameFilter filter)`.

Если каталог с указанным в конструкторе путем не существует, его можно создать логическим методом `mkdir()`. Этот метод возвращает `true`, если каталог удалось создать. Логический метод `mkdirso` создает еще и все несуществующие каталоги, указанные в пути.

Пустой каталог удаляется методом `delete()`.

Для файла можно получить его длину в байтах методом `length()`, время последней модификации в секундах с 1 января 1970 г. методом `lastModifiedo`. Если файл не существует, эти методы возвращают ноль.

Логические методы `canRead()`, `canwrite()` показывают права доступа к файлу.

Файл можно переименовать логическим методом `renameTo(Fiie newName)` или удалить логическим методом `delete o`. Эти методы возвращают `true`, если операция прошла успешно.

Если файл с указанным в конструкторе путем не существует, его можно создать логическим методом `createNewFilet`), возвращающим `true`, если файл не существовал, и `false`, если файл уже существовал.

Статическими методами `createTempFile(String prefix, String suffix, File tmpDir)` `createTempFile(String prefix, String suffix)` можно создать временный файл с именем `prefix` и расширением `suffix` в каталоге `tmpDir` или каталоге, указанном в системном свойстве `java.io.tmpdir` (см. рис. 6.2). Имя `prefix` должно содержать не менее трех символов. Если `suffix = null`, то файл получит суффикс `.tmp`.

Перечисленные методы возвращают ссылку типа File на созданный файл. Если обратиться к методу deleteOnExit(), то по завершении работы JVM временный файл будет уничтожен.

Несколько методов getxxx() возвращают имя файла, имя каталога и другие сведения о пути к файлу. Эти методы полезны в тех случаях, когда ссылка на объект класса File возвращается другими методами и нужны сведения о файле. Наконец, метод toURL() возвращает путь к файлу в форме URL.

В листинге.2 показан пример использования класса File, а на рис..4 — начало вывода этой программы.

Листинг .2. Определение свойств файла и каталога

```
import java.io.*; class FileTest{
public static void main(String[] args) throws IOException{
    PrintWriter pw = new PrintWriter( new OutputStreamWriter(System.out, "Cp866"),
true); File f = new File("FileTest.Java");
    pw.println();
    pw.println("Файл \"" + f.getName() + "\" " +(f.exists()?"":"не ") + "существует");
    pw.println("Вы " +(f.canRead()?"":"не ") + "можете читать файл");
    pw.println("Вы " +(f.canWrite()?"":"не ") + "можете записывать в файл");
    pw.println("Длина файла " + f.length() + " б"); pw.println() ;
    File d = new File(" D:\\jdk1.3\\MyProgs "); pw.println("Содержимое каталога:");
    if(d.exists() && d.isDirectory()) { String[] s = d.list();
    for(int i = 0; i < s.length; i++) pw.println(s[i]);
    }}}}
```

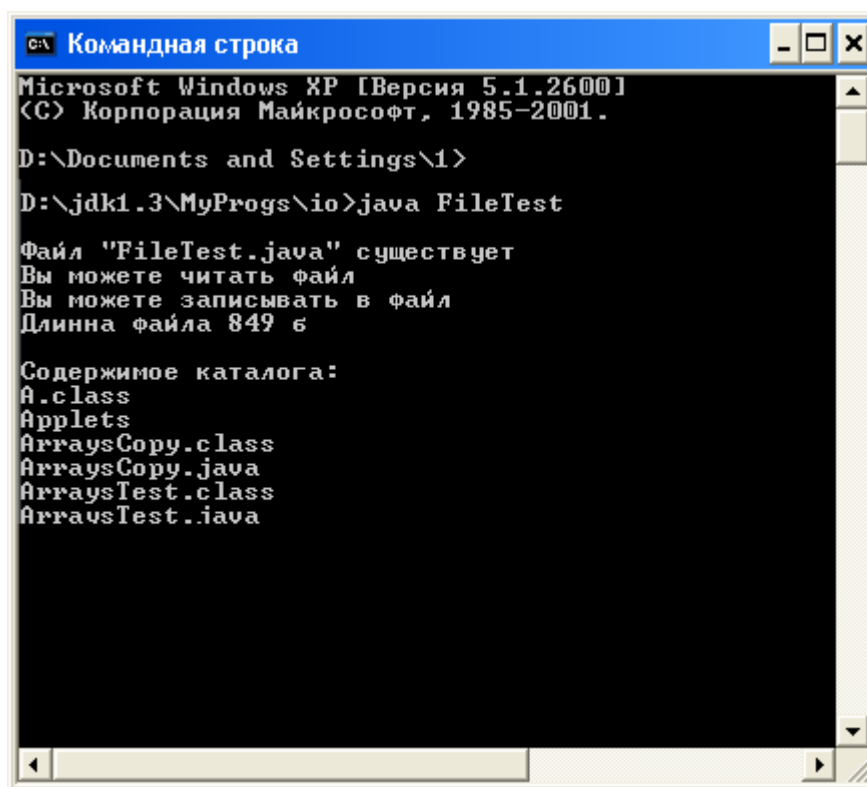


Рис. 4. Свойства файла и начало вывода каталога

Буферизованный ввод/вывод

Операции ввода/вывода по сравнению с операциями в оперативной памяти выполняются очень медленно. Для компенсации в оперативной памяти выделяется некоторая промежуточная область — буфер, в которой постепенно накапливается информация. Когда буфер заполнен, его содержимое быстро переносится процессором, буфер очищается и снова заполняется информацией.

Житейский пример буфера — почтовый ящик, в котором накапливаются письма. Мы бросаем в него письмо и уходим по своим делам, не дожидаясь приезда почтовой машины. Почтовая машина периодически очищает почтовый ящик, перенося сразу большое число писем. Представьте себе город, в котором нет почтовых ящиков, и толпа людей с письмами в руках дожидается приезда почтовой машины.

Классы файлового ввода/вывода не занимаются буферизацией. Для этой цели есть четыре специальных класса `BufferedReader`, перечисленных выше. Они присоединяются к потокам ввода/вывода как "переходное кольцо", например:

```
BufferedReader br = new BufferedReader(isr);  
BufferedWriter bw = new BufferedWriter(fw);
```

Потоки `isr` и `fw` определены выше.

Программа листинга.3 читает текстовый файл, написанный в кодировке CP866, и записывает его содержимое в файл в кодировке KOI8_R. При чтении и записи применяется буферизация. Имя исходного файла задается в командной строке параметром `args[0]`, имя копии — параметром `argstl`.

Листинг.3. Буферизованный файловый ввод/вывод

```
import java.io.*; class DOSToUNIX{  
    public static void main(String[] args) throws IOException{ if(args.length != 2){  
        System.err.println("Usage: DOSToUNIX Cp866file KOI8_Rfile"); System.exit(0);  
    }  
    BufferedReader br = new BufferedReader( new InputStreamReader(  
        new FileInputStream(args[0]), "Cp866"));  
    BufferedWriter bw = new BufferedWriter( new OutputStreamWriter(  
        new FileOutputStream(args[1]), "KOI8_R")); int c = 0;  
    while((c = br.readO) != -1)  
        bw.write((char)c); br.closeO; bw.close();  
    System.out.println("The job's finished.");  
    } }
```

Прямой доступ к файлу

Если необходимо интенсивно работать с файлом, записывая в него данные разных типов Java, изменяя их, отыскивая и читая нужную информацию, то лучше всего воспользоваться методами класса `RandomAccessFile`.

В конструкторах этого класса `RandomAccessFile(File file, String mode)` `RandomAccessFile(String fileName, String mode)` вторым аргументом `mode` задается режим открытия файла. Это может быть строка `"r"` — открытие файла только для чтения, или `"rw"` — открытие файла для чтения и записи.

Этот класс собрал все полезные методы работы с файлом. Он содержит все методы классов `DataInputStream` и `DataOutputStream`, кроме того, позволяет прочитать сразу целую строку методом `readLine()` и отыскать нужные данные в файле.

Байты файла нумеруются, начиная с 0, подобно элементам массива. Файл снабжен неявным указателем (file pointer) текущей позиции. Чтение и запись производится, начиная с текущей позиции файла. При открытии файла конструктором указатель стоит на начале файла, в позиции 0. Текущую позицию можно узнать методом `getFilePointer()`. Каждое чтение или запись перемещает указатель на длину прочитанного или записанного данного. Всегда можно переместить указатель в новую позицию, роз методом `seek(long pos)`. Метод `seek(0)` перемещает указатель на начало файла.

В классе нет методов преобразования символов в байты и обратно по кодовым таблицам, поэтому он не приспособлен для работы с кириллицей.

Пример – копирование файлов

```
// Буферизованный файловый ввод/вывод
import java.io.*; class FtoF{
public static void main(String[] args) throws IOException{
if(args.length != 2){ System.err.println("Usage: FtoF file1 file2"); System.exit(0);
}
int k;
BufferedReader br = new BufferedReader(
new InputStreamReader( new FileInputStream(args[0]) ));
BufferedWriter bw = new BufferedWriter(
new OutputStreamWriter( new FileOutputStream(args[1]) )); int c = 0;
while((c = br.read()) != -1 )
{
bw.write((char) c );
//      System.out.print( "<"+(char) c + ">");
//      System.out.print(" = ");
//      System.out.println((int) c );
}
br.close();
bw.close();
System.out.println("The job's finished.");
}
}
// Буферизованный файловый ввод/вывод с перекодировкой
```

```

import java.io.*; class WINtoKIO8{
public static void main(String[] args) throws IOException{ if(args.length != 2){
System.err.println("Usage: WINtoKIO8 Cp1251-file KOI8_R-file"); System.exit(0);
}
int k;
BufferedReader br = new BufferedReader(
new InputStreamReader( new FileInputStream(args[0]), "Cp1251")
);
BufferedWriter bw = new BufferedWriter(
new OutputStreamWriter( new FileOutputStream(args[1]), "KOI8_R")
);
int c = 0;
while((c = br.read()) != -1 ) bw.write((char) c );
br.close();
bw.close();
System.out.println("The job's finished.");
}}

```

Классы `ByteArrayInputStream` и `ByteArrayOutputStream`

Самый естественный и простой источник, откуда можно считывать байты, – это, конечно, массив байт. Класс `ByteArrayInputStream` представляет поток, считывающий данные из массива байт. Этот класс имеет конструктор, которому в качестве параметра передается массив `byte[]`. Соответственно, при вызове методов `read()` возвращаемые данные будут браться именно из этого массива. Например:

```
byte[] bytes = {1,-1,0}; ByteArrayInputStream in =
new ByteArrayInputStream(bytes);
int readedInt = in.read(); // readedInt=1
System.out.println("first element read is: "
+ readedInt);
readedInt = in.read();
// readedInt=255. Однако
// (byte)readedInt даст значение -1
System.out.println("second element read is: "+ readedInt);
readedInt = in.read();
// readedInt=0
System.out.println("third element read is: " + readedInt);
```

Если запустить такую программу, на экране отобразится следующее:

first element read is: 1 second element read is: 255 third element read is: 0

При вызове метода `read()` данные считывались из массива `bytes`, переданного в конструктор `ByteArrayInputStream`. Обратите внимание, в данном примере второе считанное значение равно 255, а не -1, как можно было бы ожидать. Чтобы понять, почему это произошло, нужно вспомнить, что метод `read` считывает `byte`, но возвращает значение `int`, полученное добавлением необходимого числа нулей (в двоичном представлении). Байт, равный -1, в двоичном представлении имеет вид 11111111 и, соответственно, число типа `int`, получаемое приставкой 24-х нулей, равно 255 (в десятичной системе). Однако если явно привести его к `byte`, получим исходное значение.

Аналогично, для записи байт в массив применяется класс `ByteArrayOutputStream`. Этот класс использует внутри себя объект `byte[]`, куда записывает данные, передаваемые при вызове методов `write()`. Чтобы получить записанные в массив данные, вызывается метод `toByteArray()`. Пример:

```
ByteArrayOutputStream out =
new ByteArrayOutputStream(); out.write(10);
out.write(11);
byte[] bytes = out.toByteArray();
```

В этом примере в результате массив `bytes` будет состоять из двух элементов: 10 и 11.

Использовать классы `ByteArrayInputStream` и `ByteArrayOutputStream` может быть

очень удобно, когда нужно проверить, что именно записывается в выходной поток. Например, при отладке и тестировании сложных процессов записи и чтения из потоков. Эти классы хороши тем, что позволяют сразу просмотреть результат и не нужно создавать ни файл, ни сетевое соединение, ни что-либо еще.

Классы `FileInputStream` и `FileOutputStream`

Класс `FileInputStream` используется для чтения данных из файла. Конструктор такого класса в качестве параметра принимает название файла, из которого будет производиться считывание. При указании строки имени файла нужно учитывать, что она будет напрямую передана операционной системе, поэтому формат имени файла и пути к нему может различаться на разных платформах. Если при вызове этого конструктора передать строку, указывающую на несуществующий файл или каталог, то будет брошено `java.io.FileNotFoundException`. Если же объект успешно создан, то при вызове его методов `read()` возвращаемые значения будут считываться из указанного файла.

Для записи байт в файл используется класс `FileOutputStream`. При создании объектов этого класса, то есть при вызовах его конструкторов, кроме имени файла, также можно указать, будут ли данные дописываться в конец файла, либо файл будет перезаписан. Если указанный файл не существует, то сразу после создания `FileOutputStream` он будет создан. При вызовах методов `write()` передаваемые значения будут записываться в этот файл. По окончании работы необходимо вызвать метод `close()`, чтобы сообщить системе, что работа по записи файла закончена. Пример:

```
byte[] bytesToWrite = { 1, 2, 3 }; byte[] bytesReaded = new byte[10]; String fileName =
"d:\\test.txt"; try {
    // Создать выходной поток
    FileOutputStream outFile = new FileOutputStream(fileName);
    System.out.println("Файл открыт для записи");
    // Записать массив
    outFile.write(bytesToWrite);
    System.out.println("Записано: " + bytesToWrite.length + " байт");
    // По окончании использования должен быть закрыт
    outFile.close();
    System.out.println("Выходной поток закрыт");
    // Создать входной поток
    FileInputStream inFile = new FileInputStream(fileName); System.out.println("Файл
открыт для чтения");
    // Узнать, сколько байт готово к считыванию
    int bytesAvailable = inFile.available(); System.out.println("Готово к считыванию: " +
bytesAvailable + " байт");
    // Считать в массив
    int count = inFile.read(bytesReaded,0,bytesAvailable); System.out.println("Считано: "
+ count + " байт"); for (i=0;i<count;i++)
    System.out.print(bytesReaded[i]+","); System.out.println();
    inFile.close();
    System.out.println("Входной поток закрыт");
} catch (FileNotFoundException e) { System.out.println("Невозможно произвести
запись в файл: " +
```



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

Пример 1.

Результатом работы программы будет:

**Файл открыт для записи Записано: 3 байт Выходной поток закрыт Файл
открыт для чтения
Готово к считыванию: 3 байт
Считано: 3 байт
1,2,3,**

Входной поток закрыт

Пример 2.

При работе с `FileInputStream` метод `available()` практически наверняка вернет длину файла, то есть число байт, сколько вообще из него можно считать. Но не стоит закладывать на это при написании программ, которые должны устойчиво работать на различных платформах,— метод `available()` возвращает число байт, которое может быть на данный момент считано без блокирования. Тот факт, что, скорее всего, это число и будет длиной файла, является всего лишь частным случаем работы на некоторых платформах.

В приведенном примере для наглядности закрытие потоков производилось сразу же после окончания их использования в основном блоке. Однако лучше закрывать потоки в `finally` блоке.

```
...  
} finally { try{inFile.close();}catch(IOExceptione){};  
}
```

Такой подход гарантирует, что поток будет закрыт и будут освобождены все связанные с ним системные ресурсы.

Сериализация объектов (serialization)

Для объектов процесс преобразования в последовательность байт и обратно организован несколько сложнее – объекты имеют различную структуру, хранят ссылки на другие объекты и т.д. Поэтому такая процедура получила специальное название - сериализация (serialization), обратное действие, – то есть воссоздание объекта из последовательности байт – десериализация.

Поскольку сериализованный объект – это последовательность байт, которую можно легко сохранить в файл, передать по сети и т.д., то и объект затем можно восстановить на любой машине, вне зависимости от того, где проводилась сериализация. Разумеется, Java позволяет не задумываться при этом о таких факторах, как, например, используемая операционная система на машине-отправителе и получателе. Такая гибкость обусловила широкое применение сериализации при создании распределенных приложений, в том числе и корпоративных (enterprise) систем.

Стандартная сериализация

Для представления объектов в виде последовательности байт определены унаследованные от `DataInput` и `DataOutput` интерфейсы `ObjectInput` и `ObjectOutput`, соответственно. В `java.io` имеются реализации этих интерфейсов – классы `ObjectInputStream` и `ObjectOutputStream`.

Эти классы используют стандартный механизм сериализации, который предлагает JVM. Для того, чтобы объект мог быть сериализован, класс, от которого он порожден, должен реализовывать интерфейс `java.io.Serializable`. В этом интерфейсе не определен ни один метод. Он нужен лишь для указания, что объекты класса могут участвовать в сериализации. При попытке сериализовать объект, не имеющий такого интерфейса, будет брошен `java.io.NotSerializableException`.

Чтобы начать сериализацию объекта, нужен выходной поток `OutputStream`, в который и будет записываться сгенерированная последовательность байт. Этот поток передается в конструктор `ObjectOutputStream`. Затем вызовом метода `writeObject()` объект сериализуется и записывается в выходной поток. Например:

```
ByteArrayOutputStream os =  
new ByteArrayOutputStream();  
Object objSave = new Integer(1); ObjectOutputStream oos =  
new ObjectOutputStream(os);  
oos.writeObject(objSave);
```

Чтобы увидеть, во что превратился объект `objSave`, можно просмотреть содержимое массива:

```
byte[] bArray = os.toByteArray();
```

А чтобы восстановить объект, его нужно десериализовать из этого массива:

```
ByteArrayInputStream is = new ByteArrayInputStream(bArray);  
ObjectInputStream ois = new ObjectInputStream(is); Object objRead =  
ois.readObject();
```

Теперь можно убедиться, что восстановленный объект идентичен исходному:

```
System.out.println("readed object is: " +  
objRead.toString()); System.out.println("Object equality is: " +  
(objSave.equals(objRead))); System.out.println("Reference equality is: " +  
(objSave==objRead));
```

Результатом выполнения приведенного выше кода будет:

readed object is: 1 Object equality is: true
Reference equality is: false

Как мы видим, восстановленный объект не совпадает с исходным (что очевидно – ведь восстановление могло происходить и на другой машине), но равен сериализованному по значению.

Как обычно, для упрощения в примере была опущена обработка ошибок. Однако, сериализация (десериализация) объектов довольно сложная процедура, поэтому возникающие сложности не всегда очевидны. Рассмотрим основные исключения, которые может генерировать метод `readObject()` класса `ObjectInputStream`.

Восстановление состояния

Итак, сериализация объекта заключается в сохранении и восстановлении состояния объекта. В Java в большинстве случаев состояние описывается значениями полей объекта. Причем, что важно, не только тех полей, которые были явно объявлены в классе, от которого порожден объект, но и унаследованных полей.

Предположим, мы бы попытались своими силами реализовать стандартный механизм сериализации. Нам передается выходной поток, в который нужно записать состояние нашего объекта. С помощью `DataOutput` интерфейса можно легко сохранить значения всех доступных полей (будем для простоты считать, что они все примитивного типа). Однако в большинстве случаев в родительских классах могут быть объявлены недоступные нам поля (например, `private`). Тем не менее, такие поля, как правило, играют важную роль в определении состояния объекта, так как они могут влиять на результат работы унаследованных методов. Как же сохранить их значения?

С другой стороны, не меньшей проблемой является восстановление объекта. Как говорилось раньше, объект может быть создан только вызовом его конструктора. У класса, от которого порожден десериализуемый объект, может быть несколько конструкторов, причем, некоторые из них, или все, могут иметь аргументы. Какой из них вызвать? Какие значения передать в качестве аргументов?

После создания объекта необходимо установить считанные значения его полей. Однако многие классы имеют специальные `set`-методы для этой цели. В таких методах могут происходить проверки, могут меняться значения вспомогательных полей. Пользоваться ли этими методами? Если их несколько, то как выбрать правильный и какие параметры ему передать? Снова возникает проблема работы с недоступными полями, полученными по наследству. Как же в стандартном механизме сериализации решены все эти вопросы?

Во-первых, рассмотрим подробнее работу с интерфейсом `Serializable`. Заметим, что класс `Object` не реализует этот интерфейс. Таким образом, существует два варианта – либо сериализуемый класс наследуется от `Serializable`-класса, либо нет. Первый вариант довольно прост. Если родительский класс уже реализовал интерфейс

Serializable, то наследникам это свойство передается автоматически, то есть все объекты, порожденные от такого класса, или любого его наследника, могут быть сериализованы.

Если же наш класс впервые реализует Serializable в своей ветке наследования, то его суперкласс должен отвечать специальному требованию – у него должен быть доступный конструктор без параметров. Именно с помощью этого конструктора будет создан десериализуемый объект и будут проинициализированы все поля, унаследованные от классов, не наследующих Serializable.

```
// Родительский класс, не реализующий Serializable public class Parent {
public String firstName;
private String lastName; public Parent(){
System.out.println("CreateParent");
firstName="old_first"; lastName="old_last"; }
public void changeNames() { firstName="new_first"; lastName="new_last"; }
public String toString() {
returnsuper.toString()+"first="+firstName+",last="+lastName;}}
// Класс Child, впервые реализовавший Serializable
public class Child extends Parent implements Serializable {
private int age; public Child(int age) {
System.out.println("CreateChild");
this.age=age; }
public String toString() {
returnsuper.toString()+"age="+age; }}
// Наследник Serializable-класса public class Child2 extends Child {
private int size;
public Child2(int age, int size) { super(age); System.out.println("Create Child2");
this.size=size; }
public String toString() {
returnsuper.toString()+"size="+size; }}
// Запускаемый класс для теста
public class Test {
public static void main(String[] arg) {
try {
FileOutputStream fos=new FileOutputStream("output.bin"); ObjectOutputStream
oos=new ObjectOutputStream(fos); Child c=new Child(2);
c.changeNames(); System.out.println(c); oos.writeObject(c); oos.writeObject(new
Child2(3, 4)); oos.close(); System.out.println("Read objects:");
FileInputStream fis=new FileInputStream("output.bin"); ObjectInputStream ois=new
ObjectInputStream(fis);
System.out.println(ois.readObject()); ois.close();
} catch (Exception e) { // упрощенная обработка для краткости
e.printStackTrace();
}}}
```

Особого внимания требуют статические поля. Поскольку они принадлежат классу, а не объекту, они не участвуют в сериализации. При восстановлении объект будет работать с таким значением static-поля, которое уже установлено для его класса в этой JVM.

1.3 Инструкция

Построчный ввод текстового файла производится с помощью метода `readLine` класса `BufferedReader`, вывод в текстовый файл выполняется методами `print`, `println` и `printf` класса `PrintWriter`.

В некоторых случаях удобнее может оказаться ввод из файла группами символов. Для этого используют поток класса `InputStreamReader` и его метод `read(char[] buf)`. При чтении заполняется символьный массив `buf`. Метод возвращает `-1`, если достигнут конец файла, или количество символов, перемещенных из файла в массив. Следует учитывать, что конец строки кодируется парой символов `'\r'` и `'\n'`.

Обобщенная схема выполнения задания такова: прочесть строку; разделить строку на элементы; обработать элементы; сформировать выходную строку; вывести строку. (В некоторых заданиях разделение строки на слова не требуется.) Эти действия повторяются, пока во входном потоке есть данные.

Слова по очереди преобразуются в числа при помощи метода `parseDouble` из класса - оболочки `Double`. Аналогичные методы есть и в классах-оболочках `Integer` и `Long`. После обработки и/или преобразования чисел они выводятся в поток методом `printf`.

1.4 Листинг программы – шаблона приведен ниже.

```
1.    import java.util.Locale;
2.    import java.io.FileReader;
3.    import java.io.BufferedReader;
4.    import java.io.PrintWriter;
5.    import java.io.IOException;
6.    public class Lab2s1 {
7.    public static void main(String[] args) throws IOException {
8.    String line;//строка, прочтенная из файла
9.    String[] Numbers;//для слов, найденных в строке
10.   double d;//результат преобразования слова в число
11.   BufferedReader input = null;
12.   PrintWriter out = null;
13.   try {
14.   input = new BufferedReader(new FileReader("matrt.txt"));
15.   out=new PrintWriter("matrt3.txt");
16.   while ((line = inputStream.readLine()) != null) {
17.   Numbers=line.split("\\s+");//разбор строки на отдельные числа
18.   for(int j=0;j<Numbers.length;j++){
19.   try{//при преобразовании может возникнуть исключение
20.   d=Double.parseDouble(Numbers[j]);
21.   /*Здесь следует добавить свой код,
22.   * обрабатывающий j-ое слово или d - результат его преобразования в
число
23.   */
24.   out.printf(Locale.ROOT,"%7.3f",d);
25.   }//try
```

```
26. catch(Exception e){} //если в число преобразуется пустая строка
27. } //for
28. out.println(); //закончить строку и перейти к новой
29. } //while
30. } //try
31. finally {
32. if (input != null) {input.close();}
33. if (out != null) {out.close();}
34. } //finally
35. }
36. }
```

1.5 Документирование кода на языке Java с использованием утилиты `javadoc`

В среду программирования Eclipse встроена поддержка утилиты `javadoc` для языка Java. Утилита `javadoc` входит в состав JDK – Java Development Kit (комплект разработчика приложений на языке Java). Она позволяет генерировать html-страницы с документацией созданных классов, если код этих классов размечен специальными комментариями (они должны начинаться со знаков `/**`, а заканчиваться `*/`).

Внутри спецкомментариев допустимы следующие специальные дескрипторы:

Таблица 1 - Список дескрипторов Javadoc

Дескриптор	Описание	Применим к
@author	Автор	класс, интерфейс
@version	Версия. Не более одного дескриптора на класс	класс, интерфейс
@since	Указывает, с какой версии доступно	класс, интерфейс, поле, метод
@see	Ссылка на другое место в документации	класс, интерфейс, поле, метод
@param	Входной параметр метода	метод
@return	Описание возвращаемого значения	метод
@exception имя класса описание @throws имя класса Описание	Описание исключения, которое может быть обработано внутри метода	метод
@deprecated	Описание устаревших блоков кода	метод
{ @link reference }	Ссылка	класс, интерфейс, поле, метод
{ @value }	Описание значения переменной	статичное поле

Например, класс `Auto` можно было бы задокументировать следующим образом:

```

/**
 * Класс Автомобиль - базовый класс для объектов транспорта
 * @author Слива М.В.
 */
public class Auto {
    /**Поле для хранения названия фирмы автомобиля */
    private String firm;
    /**Поле для хранения максимальной скорости автомобиля */
    private int maxSpeed;
    /**
     * Устанавливает значение поля { @link Auto#firm}
     * @param firma - название фирмы автомобиля */
    public void setFirm(String firma){
        firm=firma;}
    /**
     * Устанавливает значение поля { @link Auto#maxSpeed}
     * @param speed - значение максимальной скорости автомобиля */
    public void setMaxSpeed(int speed){
        maxSpeed=speed;}
    /**
     * Возвращает значение поля { @link Auto#maxSpeed}
     * @return целое значение максимальной скорости автомобиля */
    public int getMaxSpeed(){
        return maxSpeed;}
    /**
     * Возвращает значение поля { @link Auto#firm}
     * @return строку с названием фирмы автомобиля */
    public String getFirm(){
        return firm;}
    /**
     * Создает автомобиль с фирмой "Без названия" и максимальной скоростью, равной 0*/
    public Auto(){
        firm="Без названия";
        maxSpeed=0;}
    /**
     * Создает автомобиль с заданными значениями фирмы и максимальной скорости
     * @param firma - название фирмы автомобиля
     * @param speed - значение максимальной скорости автомобиля*/
    public Auto(String firma, int speed){
        firm=firma; maxSpeed=speed;}}

```


Созданные описания можно предварительно посмотреть во вкладке Javadoc нижней панели среды Eclipse (рис. 1):

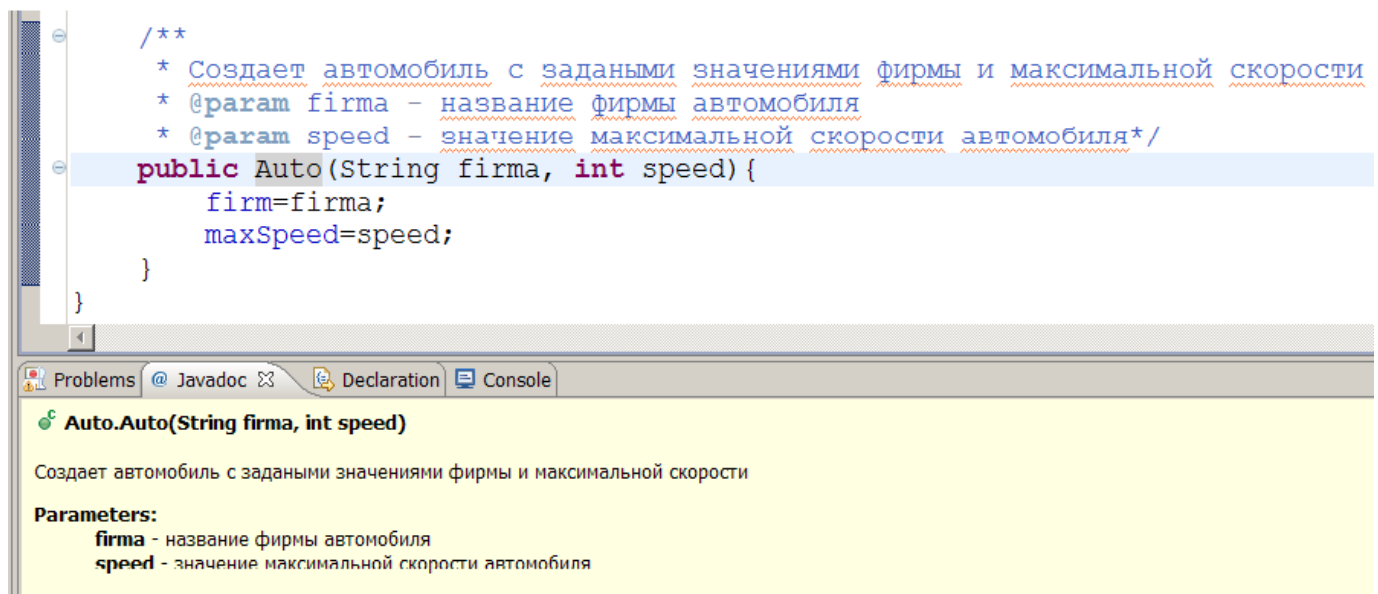


Рисунок 1 - Предварительный просмотр созданной документации

Для этого нужно просто установить курсор на названии описанного метода, поля или класса.

Чтобы сгенерировать документацию по классу в виде html-страниц, нужно выбрать команды меню Project\Generate Javadoc. Появится диалоговое окно с параметрами (рис. 2), в котором нужно выбрать путь к утилите Javadoc (как правило, это C:\Program Files\Java\jdk1.6.0_27\bin\javadoc.exe) и путь для сохранения документации (по умолчанию, это текущий проект).

После этого, нажав несколько раз Next и в конце Finish, можно получить набор html-страниц с документацией созданного класса. Каталог с ней будет примерно следующего вида (рисунок 3):

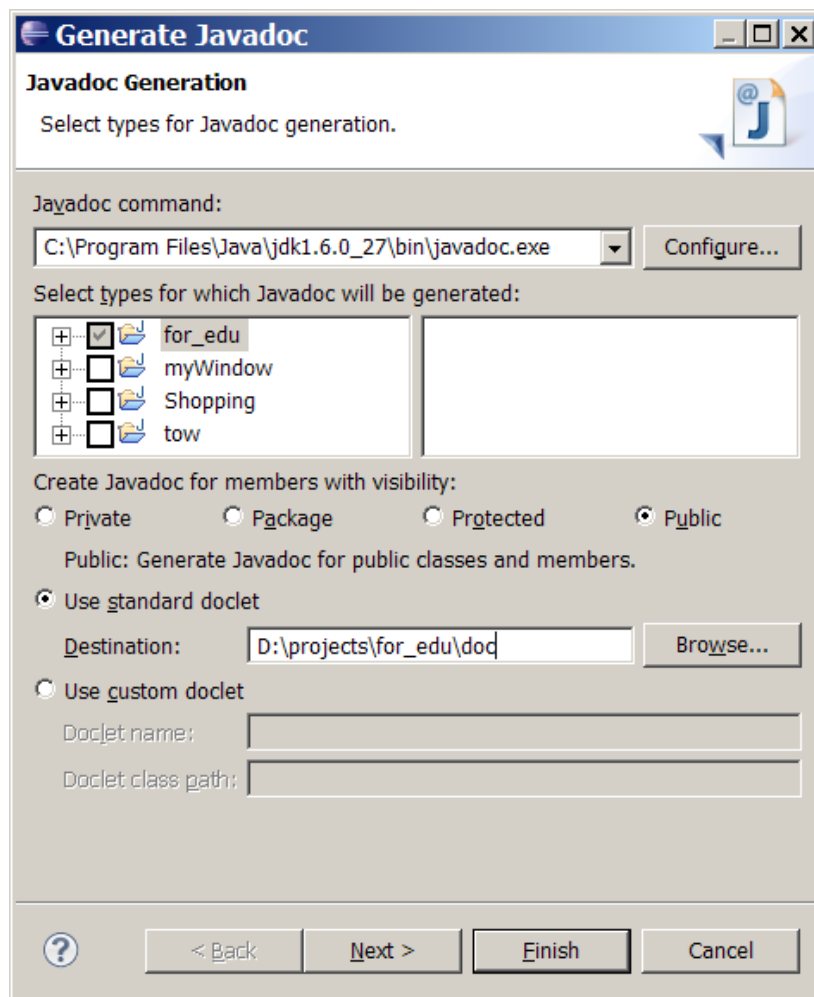


Рисунок 2 - Диалоговое окно Project\Generate Javadoc

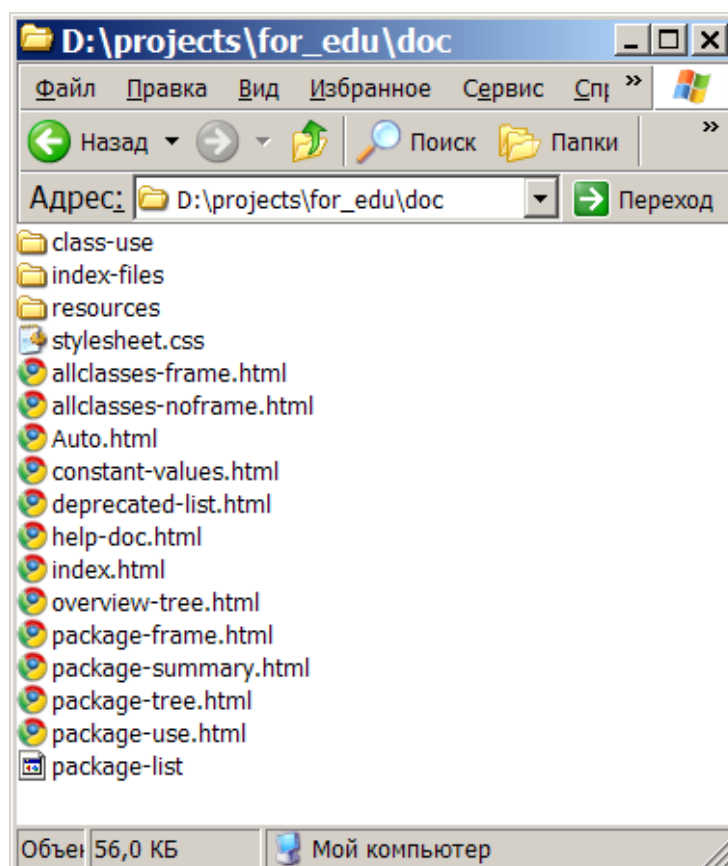


Рисунок 3 - Каталог с созданной документацией

И, запустив файл index.html, в браузере можно увидеть документацию стандартного для Java вида (рис. 4):

Class Auto

java.lang.Object
└ Auto

```
public class Auto
extends java.lang.Object
```

Класс Автомобиль - базовый класс для объектов транспорта

Author:

Слива М.В.

Constructor Summary

[Auto\(\)](#)

Создает автомобиль с фирмой "Без названия" и максимальной скоростью, равной 0

[Auto\(java.lang.String firma, int speed\)](#)

Создает автомобиль с заданными значениями фирмы и максимальной скорости

Method Summary

java.lang.String	getFirm()	Возвращает значение поля firm
int	getMaxSpeed()	Возвращает значение поля maxSpeed
void	setFirm(java.lang.String firma)	Устанавливает значение поля firm
void	setMaxSpeed(int speed)	Устанавливает значение поля maxSpeed

Рисунок 4 - Документация в сгенерированном файле

2. Задания для выполнения лабораторной работы:

2.1. Создание текстового файла

- Создать с помощью программы на Java текстовый файл **sin.txt**. Записать в файл построчно угол и значение синуса данного угла, разделенные пробелом. Углы от 0 до 360 градусов с шагом в 1 градус. Обязательно использование класса `PrintWriter`;
- Создать с помощью текстового редактора файл **input.txt**, содержащий одну строку с числом в интервале от 0 до 360.

2.2. Чтение текстового файла

- Создать двухмерный массив типа `double` (или два массива) и прочитать в него значения углов и их синусов из файла **sin.txt**;
- Прочитать значение угла из файла **input.txt** и вывести на экран значение соответствующего элемента массива.

2.3. Сериализация

- При помощи механизма сериализации сохранить созданный массив целиком в файл **sin2.dat**;
- Создать новый массив чисел типа `double` и при помощи механизма сериализации прочитать массив из файла **sin2.dat**.

2.4. Работа с содержимым файла

- Вывести в консоль, сколько строк файла имеет четную длину;
- Вывести в консоль количество чисел, у которых дробные части превышают 0,5.

2.5. Документирование кода на языке Java

- Сделать документацию по всем созданным в лабораторной работе классам.