

## **ЛАБОРАТОРНАЯ РАБОТА № 10**

**«Система ввода/вывода в java. Работа с файлами через байтовые потоки»**

**Цель:** получение навыков работы с каталогами и файлами операционной системы, а также с классами ввода/вывода, получение навыков ввода/вывода данных файла через символьные потоки.

### **Учебные вопросы:**

1. Классы пакета java.io;
2. Класс File и его методы;
3. Байтовый ввод/вывод данных;
4. Классы иерархии символьных потоков;
5. Посимвольный ввод/вывод;
6. Буферизованный ввод/вывод данных текстового файла;
7. Преобразование байтовых потоков в символьные;
8. Использование класса PrintWriter;
9. Задания для самостоятельной работы;
10. Описание результата выполнения лабораторной работы.

## 1. Классы пакета java.io

Главной задачей данной лабораторной работы является получение навыков работы с файлами (создание, запись/извлечение информации), предварительно необходимо понять принципы ввода/вывода информации и то, как файлы участвуют в этой схеме.

Иерархия основных классов системы ввода/вывода **Java** (**Java Input/Output system**) приведена на рисунке 1.

**File** – единственный класс в **java.io**, который работает непосредственно с дисковыми файлами. При этом каталог в **Java** трактуется как обычный файл, но с дополнительными свойствами.

**Остальные классы** пакета обеспечивают ввод и вывод данных, базируясь на потоках.

**Поток** является абстракцией, которая или производит, или потребляет информацию. Потоки связываются с различными физическими устройствами, но при этом ведут себя одинаково во всех случаях. Таким образом, одни и те же классы и методы ввода/вывода можно применять к устройствам любого типа. Это означает, что поток ввода может извлекать много различных видов входных данных из файла на диске, с клавиатуры или сетевого ресурса. Аналогично, поток вывода может обратиться к консоли, дисковому файлу или сетевому соединению (сокету).

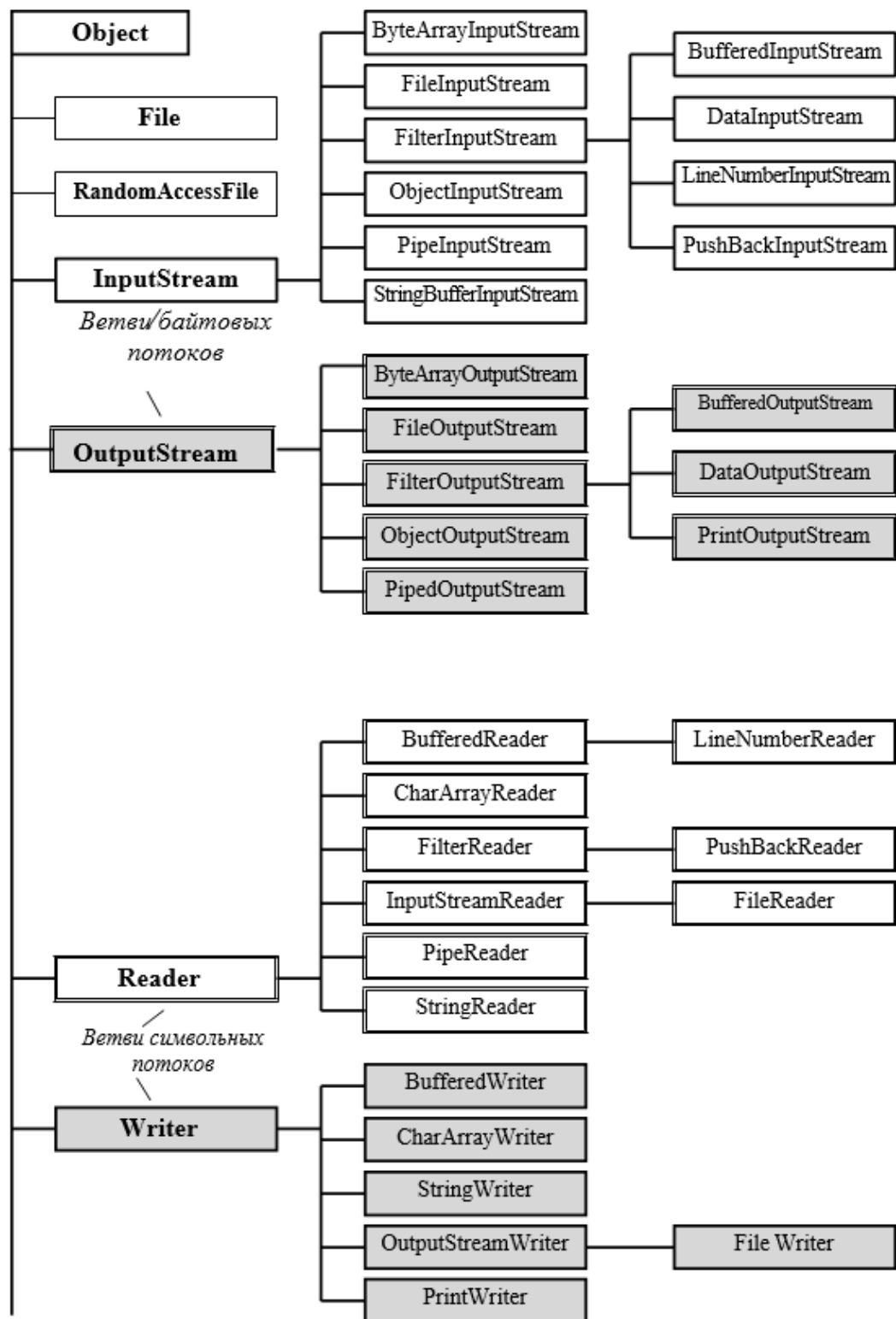


Рисунок 1

**Java** определяет два типа потоков: байтовый и символьный.

**Байтовые потоки** предоставляют удобные средства для обработки ввода и вывода байт. Они определяются в двух иерархиях классов, вершинами которых являются абстрактные классы:

- **InputStream** – для ввода (чтения) данных имеет метод **read()**;
- **OutputStream** – для вывода (записи) данных имеет метод **write()**.

**Символьные потоки** предоставляют удобные средства для обработки ввода и вывода символов. Они позволяют работать с кодировкой **Unicode**, в которой символы представляются двухбайтным кодом. Байтовые потоки этого делать не позволяют. Они зачастую работают с текстом упрощенно – просто отбрасывают старший байт каждого символа.

Символьные потоки определяются в двух других иерархиях классов, вершинами которых являются абстрактные классы:

- **Reader** – для ввода (чтения) данных имеет метод **read()**;
- **Writer** – для вывода (записи) данных имеет метод **write()**.

В иерархии классом **java.io** следует уделить внимание отдельной ветви, начинающейся с класса **RandomAccessFile**. Она обеспечивает произвольный доступ к содержимому при различных операциях.

## 2. Класс **File** и его методы

Для работы с файлами и каталогами как объектами файловой системы в **java** имеется класс **File**. Он позволяет создавать, переименовывать, удалять файлы и каталоги, проверять, существует ли файл или каталог с определенным именем, сравнивать файлы, получать информацию о файлах и др.

Конструкторы класса **File**:

- **File(String path)** – создает объект **File** на основе строки **path**, содержащей полное имя файла;
- **File(String dirName, String name)** – создает объект **File** с именем **name**, который размещен в каталоге **dirName**. Если **dirName** содержит пустую строку, считается, что файл **name** должен располагаться в каталоге, предусмотриваемом по умолчанию значениями настроек операционной системы;
- **File(File fileDir, String name)** – создает объект **File**, представляющий файл с именем **name**; файл размещен в каталоге, наименование которого определяется объектом **fileDir** типа **File**.

Основные методы данного класса представлены в таблице 1.

Ниже приведем пример работы с этими методами, используемыми для создания файлов и каталогов. Команды, где создается файл, обязательно надо

помещать в блок **try-catch** для реагирования на возникновение исключительных ситуаций.

### Пример 1. Создание файлов и папок.

```
public class KlassFile1 {
public static void main(String[] args) {
    try {
        // Создание файла в текущей папке (где расположен файл KlassFile1.java)
        File f1=new File("MyFile1.txt");
        f1.createNewFile();
        if (f1.exists()) {
            System.out.println("Создан!!!!");
            System.out.println("Полный путь1:  "+ f1.getAbsolutePath());
        }

        // Создание файла на диске E и вывод полного пути
        File f2=new File("E:\\MyFile2.txt");
        f2.createNewFile();
        System.out.println( "Полный путь 2:  "+ f2.getAbsolutePath());

        // Создание нескольких вложенных папок
        File f3=new File("E:\\Папка1\\Папка2\\Папка3");
        f3.mkdirs();
        System.out.println("Полный путь 3:  "+ f3.getAbsolutePath());
    } catch (Exception e) {
        System.out.println("Ошибка!!! "+e);
    }
}}
```




Таблица 1

Название метода	Тип	Выполняемые действия
Методы Get: <b>getName()</b> , <b>getParent()</b> , <b>getPath()</b> , <b>getAbsolutePath ()</b> , <b>getCanonicalPath()</b>	String	Позволяют получить информацию о компонентах полного имени файла или альтернативные варианты его представления
<b>Exists();</b>	boolean	Возвращает <b>true</b> , если файл существует в файловой системе
<b>canRead;</b>	boolean	Возвращает <b>true</b> , если файл существует и его содержимое может быть считано
<b>canWrite</b>	boolean	Возвращает <b>true</b> , если файл существует и его содержимое может быть прочитанным
<b>isFile;</b>	boolean	Возвращает <b>true</b> , если файл не является каталогом
<b>isDirectory;</b>	boolean	Возвращает <b>true</b> , если файл является каталогом
<b>isAbsolute;</b>	boolean	Возвращает <b>true</b> , если файл адресуется с помощью абсолютного имени
<b>isHidden;</b>	boolean	Возвращает <b>true</b> , если файл является скрытым
<b>compareTo</b>		Два имени файла могут сопоставляться в лексикографическом порядке: метод возвращает отрицательное, нулевое или положительное числовые значения
<b>lastModified()</b>	long	Возвращает ненулевое значение типа <b>long</b> , представляющее момент времени, когда файл подвергался изменению в последний раз, либо нуль, если файл не существует

Название метода	Тип	Выполняемые действия
<b>long length()</b>	long	Возвращает длину файла в байтах либо нуль, если файл не существует
<b>renameTo(File newName)</b>	boolean	Переименовывает файл, возвращая значение <b>true</b> , если результат операции успешен
<b>delete()</b>	boolean	Удаляет файл или каталог, задаваемый текущим объектом <b>File</b> , и возвращает значение <b>true</b> при успешном завершении операции. Каталог, подлежащий удалению, должен быть пустым
<b>createNewFile()</b>	boolean	Создает новый пустой файл с именем, определяемым текущим объектом <b>File</b> . Возвращает значение <b>false</b> , если файл уже существует либо не может быть создан
<b>mkdir()</b>	boolean	Создает каталог с именем, определяемым текущим объектом <b>File</b> , и возвращает значение <b>true</b> при успешном завершении операции
<b>mkdirs()</b>	boolean	Создает иерархию каталогов, перечисленных в строке полного имени, которое задается текущим объектом <b>File</b> , и возвращает <b>true</b> , если все каталоги были успешно созданы. Другими словами, некоторые каталоги могут быть созданы даже в том случае, если общий результат операции равен <b>false</b>

### 3. Байтовый ввод/вывод данных

Работая с вводом/выводом данных, необходимо постоянно контролировать обращение к классам по схеме наследования (см. рис. 1) и создавать экземпляры классов следует с учетом устройств, откуда будет поступать информация в поток, с учетом типа данных, с учетом требований к буферизации и т.д.

Знакомство с системой ввода-вывода следует начинать с ветвей байтовых потоков, классы которых изначально были представлены в стандартной системе ввода-вывода **Java** первых версий. Ветви символьных потоков появились начиная с 5 версии (jdk5.1 и др.).

В табл. 2 сделаны пояснения по некоторым классам иерархии байтовых потоков, используемых в данной лабораторной работе.

Таблица 2

Поточный класс	Назначение
InputStream OutputStream	Абстрактные классы, которые описывают поточный ввод и вывод
BufferedInputStream BufferedOutputStream	Классы буферизированных потоков ввода и вывода
ByteArrayInputStream ByteArrayOutputStream	Класс потока ввода, который читает из массива типа byte Класс потока вывода, который записывает в массив типа byte
FileInputStream FileOutputStream	Класс потока ввода, который читает из файла Класс потока вывода, который записывает в файл
DataInputStream DataOutputStream	Классы потоков, позволяющие осуществлять ввод/вывод данных, указанных стандартных примитивных типов (int, double, boolean и т.д.). Используются для чтения методы <b>ReadInt()</b> , <b>ReadDouble()</b> , ..., для записи: <b>WriteInt()</b> , <b>WriteDouble()</b> и др. в соответствии с необходимым типом данных
ObjectInputStream ObjectOutputStream	Классы потоков, которые описывают поточный ввод и вывод объектов

При работе с файлами необходимо обрабатывать ошибки, генерируемые, прежде всего, методами **read()** и **write()**.

**Общий алгоритм работы с потоками чтения (классы ветви **InputStream**):**



1. Открыть поток для чтения и связать его с устройством.

2. Пока не конец файла:

а) читать побайтно (или буферами) данные потока (метод **read()** или его варианты);

б) обрабатывать данные.

3. Заккрыть поток (метод **close()**).

**Общий алгоритм работы с потоками записи** (классы ветви **OutputStream**) при известном количестве данных:

1. Открыть поток для записи и связать его с устройством.

2. Ввести количество данных (строк, чисел или др.) – n.

3. Для номера от 0-го до n-1-го значения:

а) получить (рассчитать, сгенерировать, ввести с клавиатуры и т.д.) данные в соответствии с заданием;

б) записать побайтно (или буферами) данные в поток (метод **write()** или его варианты).

4. Дописать остаток данных на диск (метод **flush()**).

5. Заккрыть поток (метод **close()**).

В случае неизвестного количества данных при записи следует пользоваться циклом **while**, задав условием выхода ввод какого-либо значения, например 0.

**Пример 2.** Прочитать и вывести на экран информацию из трех источников: файла на диске, интернет-страницы и массива типа **byte**.

```
public class Primer1 {
```

```
    // Метод для чтения данных из потока по байтам с выводом
```

```
    public static void readAllByByte(InputStream in) throws IOException {
        while (true) {
            int oneByte = in.read(); // читает 1 байт
            if (oneByte != -1) {      // признак отсутствия конца файла
                System.out.print((char) oneByte);
            } else {
                System.out.print("\n" + "end");
                break;
            }
        }
    }
```

```
    public static void main(String[] args) throws IOException {
        try {
```

```
            // С потоком связан файл
```

```
            InputStream inFile = new FileInputStream("c:/tmp/text.txt");
            readAllByByte(inFile);
            System.out.print("\n\n\n");
            inFile.close();
```

Файл предварительно создан  
и заполнен данными

```
            // С потоком связана интернет-страница
```

```
            InputStream inUrl = new URL("http://google.com").openStream();
            readAllByByte(inUrl);
            System.out.print("\n\n\n");
            inUrl.close();
```

```
            // С потоком связан массив типа byte
```

```
            InputStream inArray=new ByteArrayInputStream(new byte [] {7, 9, 3,7,4});
            readAllByByte(inArray);
            System.out.print("\n\n\n");
            inArray.close();
```

массив

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

**Пример 3.** Прочитать и вывести на экран информацию из предварительно созданного файла с использованием буфера в 5 байт.

```
public class File_ByteRead_SamBuff {
```

```
// Считывание по 5 символов (буфер)
```

```
public static void readAllByArray(InputStream in) throws IOException {
```

```
    byte [] buff = new byte[5];
```

```
    while (true) {
```

```
        int count = in.read(buff);
```

```
        if (count != -1) { // если не конец файла
```

```
            System.out.println("количество = " + count + ", buff = "
```

```
                                + Arrays.toString(buff) + ", str = "
```

```
                                + new String(buff, 0, count, "cp1251" )); //"UTF8"
```

```
        } else {
```

```
            break;
```

```
        }
```

```
    }
```

```
}
```

```
public static void main(String[] args) throws IOException {
```

```
    String fileName = "E:\\MyFile1.txt";
```

```
    InputStream inFile = null;
```

*← переменная объявляется до секции try, чтобы не ограничивать область видимости*

```
    try {
```

```
        inFile = new FileInputStream(fileName);
```

```
        readAllyByArray(inFile);
```

```
    } catch (IOException e) {
```

```
        System.out.println("Ошибка открытия-закрытия файла " + filename+e);
```

```
    } finally {
```

*// корректное закрытие потока*

```
        if (inFile != null) {
```

```
            try {
```

```
                inFile.close();
```

```
            } catch (IOException ignore) {
```

```
                /*NOP*/
```

```
            }
```

```
    } }
```

*← «No OPeration» – ничего не делать*

Следует отметить, что при работе с байтовыми потоками существует возможность читать и записывать данные точно определенного типа. Для этой цели необходимо использовать классы **DataInputStream** и **DataOutputStream** с их методами для чтения **ReadInt()**, **ReadDouble()** и др. и для записи – **WriteInt()**, **WriteDouble()** и др. Для строк в формате UTF-8, где символы кодируются одним байтом, используются методы **readUTF()** и **writeUTF()**.

**Пример 4.** Создать первый файл на диске и записать в него заданное количество вещественных чисел. Далее создать второй файл и переписать в него все числа из первого файла.

Программу составить в соответствии со следующим алгоритмом:

1. Создать новую папку **My** на диске.
2. В папке создать 1-й файл *numIsh.txt*.
3. В файл записать через поток числа, вводимые с клавиатуры.
4. В той же папке создать 2-й файл *numRez.txt*.
5. Последовательно считывая числа из 1-го файла *numIsh.txt*, переписать их во второй файл *numRez.txt*.
6. Закончить.

Более подробно распишем пункт 5 (чтение-запись):

5.1) открыть 1-й поток для чтения из 1-го файла *numIsh.txt*; 5.2) открыть 2-й поток для записи во 2-й файл *numRez.txt*; 5.3) пока не конец 1-го файла:

- а) считать число из 1-го потока (файла *numIsh.txt*);
- б) записать число во 2-й поток (файл *numRez.txt*);
- в) вывести число на экран;

5.4) дописать остаток данных на диск; 5.5) закрыть 1-й поток;

5.6) закрыть 2-й поток.

Ниже приведен код программы.

```

public class FilesData {
    public static void main(String[] args) {
        try{
            // Создание исходного файла numIsh.txt и запись в него чисел типа float

            File f1=new
File("E:\\My\\numIsh.txt");
f1.createNewFile();

            Scanner sc = new Scanner(System.in, "cp1251");

            DataOutputStream wr =
new DataOutputStream(new FileOutputStream(f1.getAbsolutePath()));
            System.out.println("Сколько вещественных чисел записать в
файл?"); int count = sc.nextInt();

            System.out.println("Введите числа:");
            for (int i = 0; i < count; i++)
                wr.writeFloat(sc.nextFloat());

            wr.flush();
            wr.close();

            // Создание файла numRez.txt и переписывание в него чисел из numIsh.txt

            File f2=new
File("E:\\My\\numRez.txt");
f2.createNewFile();

            // поток для чтения из исходного файла numIsh.txt
            DataInputStream rd =
new DataInputStream(new FileInputStream(f1.getAbsolutePath()));
            // поток для записи в результирующий файл numRez.txt
            wr = new DataOutputStream(new FileOutputStream(f2.getAbsolutePath()));

            try{
                while(true){
                    float number=rd.readFloat();
                    wr.writeFloat(number);
                    System.out.println(" Число "+ (float)number);
                }
            }catch(EOFException e){ }
        }
    }
}

```

} чтение-запись из одного  
файла в другой

```

        wr.flush();
        wr.close();
        rd.close();
    }catch(IOException e){
System.out.println("End of file");
    }
    }}

```

**Пример 5.** Создать файл на диске, ввести заданное с клавиатуры количество строк текста и записать их в файл в формате UTF-8.

```

public class Files_byteRW_my2 {

    public static void main(String[] args) {
Scanner sc= new Scanner(System.in);
System.out.print("Введите имя файла => "); String
fname=sc.nextLine();
    try{
        // Создается файл
        File f1=new File(fname);
f1.createNewFile();           // файл создан
        System.out.println("Полный путь файла:  "+ f1.getAbsolutePath());

        System.out.print("Введите количество строк для записи в файл => ");
int n=sc.nextInt();

        // Создается поток для записи с учетом типа данных – DataOutputStream,
// и ему в качестве параметра передается поток FileOutputStream
        DataOutputStream dOut=
new DataOutputStream( new FileOutputStream(f1));
        sc.nextLine();    //очистка буфера
        for (int i = 0; i < n; i++) {
            System.out.print("Введите строку для записи в файл => ");
String s=sc.nextLine();
            dOut.writeUTF(s );
            //или dOut.writeUTF(s + "\n" ); – запись отдельных строк
        }
        dOut.flush();    // дописываем несохраненные данные на диск
        dOut.close(); // закрываем поток
    }
}

```

```
// Чтение и вывод данных из созданного файла
```

```
    DataInputStream dIn=new DataInputStream(new FileInputStream(f1));  
while (true) {  
    System.out.println(rd.readUTF());  
    }  
    }catch (Exception e) {  
System.out.println(""+e);  
    }  
    }  
    }
```

Классы ***DataInputStream*** и ***DataOutputStream*** при работе с числовыми типами данных и с данными булевского типа более предпочтительны по сравнению с другими классами библиотеки ввода/вывода **java**, но при работе с обычными строками они не применяются. Например, при попытке считать строку типа String среда выдаст предупреждающую запись в виде:

```
String s= dIn.readLine();
```

#### 4. Классы иерархии символьных потоков

Потоки, ориентированные на байтовые последовательности, не- удобны для обработки информации, записанной в формате **Unicode**, где один символ записан двумя байтами. Именно с учетом этой особенности для обработки символов используются классы, выделенные в отдельную иерархию, – это **Reader** и **Writer** (рис. 1, 2). Их основные классы-наследники описаны в табл. 3.

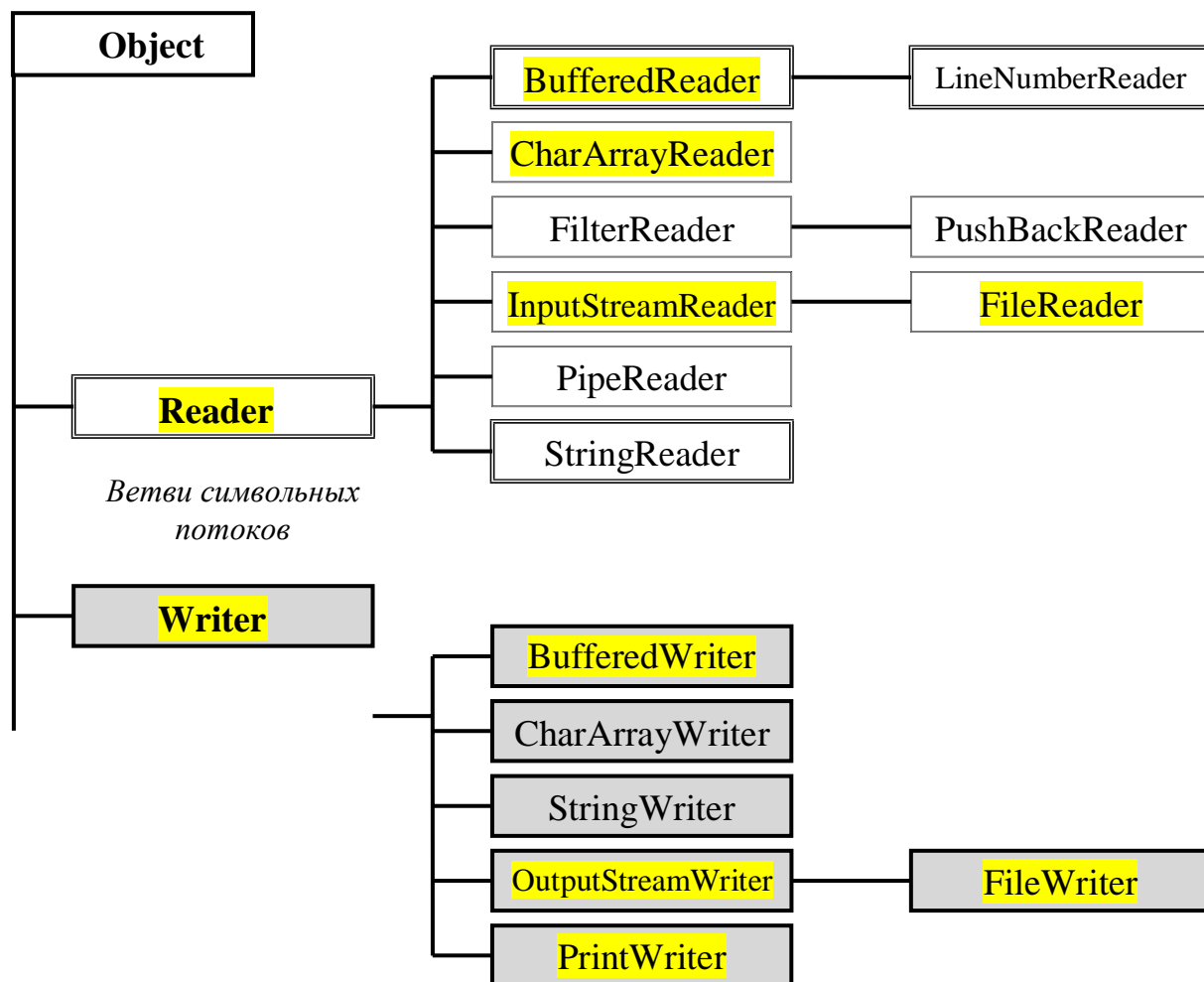


Рис. 2



Таблица 3

Символьные классы	Назначение
BufferedReader/ BufferedWriter	<p>Буферизированные символьные потоки ввода/вывода, позволяющие обрабатывать целые строки. Имеют методы: <b>readLine()/writeLine()</b></p> <p>Принимают в качестве параметров: символьный поток и размер буфера.</p> <p>Если буфер не указан – используется буфер по умолчанию 8192 байта</p>
FileReader / FileWriter	<p>Поток ввода/вывода символов файла.</p> <p>Принимают в качестве параметров: файл или путь к файлу.</p> <p>Кроме того, есть конструктор <b>FileWriter(String fileName, boolean append)</b>, где параметр <b>append=true</b> позволяет использовать файл на добавление информации.</p> <p>Требует обязательной обработки исключительной ситуации <b>FileNotFoundException</b></p>
CharArrayReader/ CharArrayWriter	Потоки ввода/вывода символов, принимающие в качестве параметров массивы символов
StringReader/ StringWriter	Поток ввода/вывода указанных строк. StringReader принимает строку в качестве параметра
InputStreamReader/ OutputStreamWriter	Конвертируют потоки <b>InputStream/OutputStream</b> и их наследников в символьные потоки, позволяя задать нужную кодировку текста
PrintWriter	<p>Поток вывода, который поддерживает методы <b>print()</b> и <b>println()</b>. Работает быстрее, чем <b>System.out.println()</b>.</p> <p>Конструкторы класса:</p> <p><b>PrintWriter(OutputStream out);</b></p> <p><b>PrintWriter(OutputStream out, boolean autoFlush);</b></p> <p>Второй конструктор создает поток с автоматическим сбросом (очисткой) буфера для параметра <b>autoFlush=true</b></p>

Следует помнить, что экземпляры абстрактных классов **Reader** и **Writer** создавать нельзя, а можно только объявлять переменные данного типа. Создаются экземпляры классов, ориентированных на определенные источники информации (файлы, массивы символов и т.д.).

Из классов библиотеки ввода/вывода, реализующих потоки, можно составить множество разнообразных конфигураций. Далее приведены примеры наиболее употребляемых конструкций (типичных сочетаний классов) для символьных потоков, встречающихся при решении практических задач. В примерах используется упрощенная обработка исключений.

Все потоки после обработки следует закрывать методом **close()**.

## 5. Посимвольный ввод/вывод

Самая простая, но неэффективная (медленная) реализация чтения/записи данных файла может быть продемонстрирована с использованием классов **Reader – FileReader**, **Writer – FileWriter**.

Основные методы:

–**read()** для чтения;

–**write()** для записи с удалением ранее имеющейся информации в файле;

–**append()** для добавления данных в файл (существующая ранее информация в файле не уничтожается).

**Пример 6.** Чтение из одного файла и запись в другой файл данных посимвольно.

```
public class  
File_RW_byByte {
```

*Метод main генерирует исключение*

```
    public static void main(String[] args) throws IOException {  
        Reader in=null;           // можно сразу запустить FileReader in=null;  
        Writer out=null;          // можно сразу запустить FileWriter out =null;  
        try {  
            in = new FileReader("E:\\MyFile1.txt");           // файл для чтения  
            out= new FileWriter("E:\\MyFile2.txt", true);      // файл для записи
```

*разрешено добавление*

*// Данные считываются и записываются побайтно, как и для*

*// InputStream/OutputStream*

`int oneByte;` *// переменная, в которую считываются данные*

`while ((oneByte = in.read()) != -1) {`

`// out.write((char)oneByte);` *// запись с уничтожением ранее*

*// существующих данных*

`out.append((char)oneByte);` *// запись с добавлением данных в конец*

`System.out.print((char)oneByte);`

`}`

`} catch (IOException e) {`

`System.out.println("Ошибка!!!! ");`

`}`

`finally{`

`in.close();`

`out.close();`

`} }`

## 6. Буферизованный ввод/вывод данных текстового файла

Значительно ускорить процесс чтения/записи помогает буферизация ввода/вывода, для этого полученная ссылка на экземпляр класса **FileReader/FileWriter** передается в качестве параметра в конструкторы класса **BufferedReader/BufferedWriter**.

Ниже приведены примеры записи возможных конструкций при создании объектов.

*Размер буфера по умолчанию 8192 байта*

`BufferedReader br1 = new BufferedReader( new FileReader("E:\\ File1.txt" ));`

`BufferedReader br1 = new BufferedReader( new FileReader("E:\\ File1.txt" ), 1024);`

`BufferedWriter out = new BufferedWriter( new FileWriter("E:\\ File2.txt" ));`

Недостаток вышеприведенных конструкций — нет возможности управлять кодировкой.

Класс **BufferedReader** позволяет использовать метод **readLine()** для чтения отдельных строк. При достижении конца файла метод **readLine()** возвращает ссылку **null**.

Метод **writeLine()** класса **BufferedWriter** позволяет производить построчную запись.

При буферизированной записи данных в файл перед закрытием потока следует выполнять операцию очистки (сбрасыванием) буфера с дописыванием данных на диск **flush()**.

**Пример 7.** Чтение из одного файла и запись в другой файл данных построчно с использованием буфера в 1 килобайт.

```

public class Buf_RW_3 {
    public static void main(String[] args) throws IOException{
        BufferedReader br = null;
        BufferedWriter out=null;
        try {
            // Создание файловых символьных потоков для чтения и записи
            br = new BufferedReader( new FileReader("E:\\MyFile1.txt" ), 1024);
            out = new BufferedWriter( new FileWriter( "E:\\MyFile2.txt" ));


            int lineCount = 0;          // счетчик строк
            String s;

            // Переписывание информации из одного файла в другой
            while ((s = br.readLine()) != null) {
                lineCount++;
                System.out.println(lineCount + ": " + s);
                out.write(s);
                out.newLine();          // переход на новую строку
            }

        } catch (IOException e) {
            System.out.println("Ошибка !!!!!!!");
        }
        finally{
            br.close();
            out.flush();
            out.close();
        }
    }
}

```

размер буфера



## 7. Преобразование байтовых потоков в символьные

В некоторых ситуациях для решения задачи используются как «байтовые», так и «символьные» классы. Для этого в библиотеке имеются классы-адаптеры: ***InputStreamReader*** – конвертирует ***InputStream*** в ***Reader***, а ***OutputStreamWriter*** – трансформирует ***OutputStream*** в ***Writer***. При этом возможна передача в качестве второго параметра нужной кодовой страницы, позволяющей выводить текст в надлежащем виде.

Конструкторы с кодировкой:

**InputStreamReader**(<поток для чтения>, "<кодировка>");

**OutputStreamWriter**(<поток для записи>, "<кодировка>").

**Пример 8.** Прочитать и вывести на экран информацию из трех источников: файла на диске, интернет-страницы и массива данных типа **byte**. Указать кодировку, поддерживающую кириллицу. *(Сравнить с работой программы, приведенной в примере 2.)*

```
public class InConvertInText {

    public static void readAllByByte( Reader in)          throws IOException {
        while (true) {
            int oneByte = in.read();                      // читает 1 байт
            if (oneByte != -1) {                            // признак конца файла
                System.out.print((char) oneByte);
            } else {
                System.out.print("\n" + " конец ");
                break;
            }
        }
    }
}
```

```


public static void main(String[] args) {
    try {
        // С потоком связан файл
        InputStream inFile = new FileInputStream("E:\\MyFile1.txt");    // байтовый
                                                                    // поток
        Reader rFile= new InputStreamReader(inFile,"cp1251");    // символный
                                                                    // поток
        readAllByByte(rFile);
        System.out.print("\n\n\n");
        inFile.close();
        rFile.close();
        // С потоком связана интернет-страница

        InputStream inUrl = new URL("http://google.com").openStream(); // байтовый
                                                                    // поток
        Reader rUrl=new InputStreamReader(inUrl, "cp1251");    // символный
                                                                    // поток

        readAllByByte(rUrl);
        System.out.print("\n\n\n");
        inUrl.close();
        rUrl.close();
        // С потоком связан массив типа byte
        InputStream inArray = new ByteArrayInputStream( new byte[] {5, 8, 3, 9, 11});
        Reader rArray=new InputStreamReader(inArray,"cp1251" ); // символный
                                                                    // поток

        readAllByByte(rArray);
        System.out.print("\n\n\n");
        inArray.close();
        rArray.close();
    } catch (IOException e) {
        System.out.println("Ошибка: "+ e);
    }
}
}

```

передается «русская»  
кодировка
 

Наиболее быстро и корректно работают буферизированные символьные потоки, построенные на байтовых потоках. Конструкция строится с использованием трех классов, как показано в нижеприведенном примере.



**Пример 9.** Чтение из одного файла и запись в другой файл данных построчно с использованием буферизации символьных потоков основанных на байтовых файловых потоках.

```
public class Buf_WR_IO_4 {  
    public static void main(String[] args) throws IOException {  
        BufferedReader br = null;  
        BufferedWriter bw=null;  
        try {  
            // Создание потоков для чтения и записи с нужной кодировкой  
            br = new BufferedReader(  
                new InputStreamReader(  
                    new FileInputStream("E:\\MyFile1.txt"),"cp1251"));  
  
            bw = new BufferedWriter(  
                new OutputStreamWriter(  
                    new FileOutputStream("E:\\MyFile2.txt"),"cp1251"));  
  
            // Переписывание информации из одного файла в другой  
            int lineCount = 0; // счетчик строк  
            String s;  
            while ((s = br.readLine()) != null) { lineCount++;  
                System.out.println(lineCount + ": " + s);  
                bw.write(lineCount + ": " + s); // запись без перевода строки  
                bw.newLine(); // принудительный переход на новую строку  
            }  
        } catch (IOException e) {  
            System.out.println("Ошибка!!!!!!!!!!");  
        }  
        finally{  
            br.close();  
            bw.flush();  
            bw.close();  
        }  
    }  
}
```

## 8. Использование класса **PrintWriter**

Для записи или вывода на консоль очень удобно использовать класс **PrintWriter**. Он дает возможность указать требуемую кодировку, имеет методы **print()** и **println()** для записи строк без перехода на новую строку или с переходом.

**PrintWriter** позволяет в качестве параметра принимать выходной поток **System.out** и осуществлять вывод на консоль. При этом он работает намного быстрее, чем **System.out.println()**.

**Пример 10.** Выполнить чтение из одного файла и запись в другой файл с использованием класса **PrintWriter**.

```
public class Buf_RW_2 {  
    public static void main(String[] args) {  
        BufferedReader br = null;  
        PrintWriter out=null;  
        try {  
            // Создание потоков  
            br = new BufferedReader(  
                new InputStreamReader(  
                    new FileInputStream("E:\\MyFile1.txt"),"cp1251"));  
  
            PrintWriter out = new PrintWriter("E:\\MyFile2.txt","cp1251");  
            // Переписывание информации из одного файла в другой  
            int lineCount = 0;  
            String s;  
            while ((s = br.readLine()) != null) {  
                lineCount++;  
                out.println(lineCount + ": " + s);  
            }  
        } catch (IOException e) {  
            System.out.println("Ошибка !!!!!!!");    }  
        finally{  
            br.close();  
            out.flush();  
            out.close();  
        }  
    }  
}
```

Представленный ниже фрагмент кода демонстрирует работу **PrintWriter** с системным выходным потоком:

...

```
PrintWriter out = new PrintWriter(System.out);
```

```
int lineCount = 0;
```

```
String s;
```

```
// Вывод информации из файла на монитор
```

```
while ((s = br.readLine()) != null)
```

```
{
```

```
lineCount++;
```

```
out.println(lineCount + ": " + s);
```

```
}
```

...

## 9. Задания для самостоятельной работы

**Задание 1.** В отдельных проектах выполнить примеры 1 – 10 лабораторной работы. Протестировать программы с помощью отладчика. Выявить различие в работе программ в примерах 7 и 8.

**Задание 2.** Создать проект, позволяющий из одного, предварительно созданного программными средствами файла, переписать данные, соответствующие условию - в исходном файле содержится две строки в формате UTF-8 и 5 чисел типа double. В результирующий файл переписать вторую строку и положительные числа.

**Задание 3.** Создать проект, позволяющий из одного текстового файла, содержащего несколько строк (тип String) заранее подготовленного текста на русском языке (Пушкин, Лермонтов или другой российский классик на Ваш вкус), построчно переписать в другой текстовый файл слова начинающиеся с согласных букв..

Требования:

- слова из предложения выделять методом **split()**;
- в новом файле следует указать номер строки, в которой искомые слова находились в исходном файле;
- для каждой строки указать количество выбранных слов.

## 10 Описание результата выполнения лабораторной работы

В отчете по лабораторной работе должны быть представлены все примеры и задания.

Лабораторная работа принимается при наличии отчета всех выполненных заданий.

Структура отчета по лабораторной работе:

1. Титульный лист;
2. Цель работы;
3. Описание задачи;
4. Ход выполнения (содержит код программы);
5. Вывод;

Оформление:

- а) шрифт Times New Roman;
- б) размер шрифта 12 или 14;
- в) межстрочный интервал 1,5.

Отчет выполняется индивидуально и направляется по адресу электронной почты [proverkalab@yandex.ru](mailto:proverkalab@yandex.ru). В теле письма необходимо указать ФИО студента и номер группы.