**Лабораторная работа №2**

**«Работа со строками в Java. Потоки данных»**

**Цель работы:**

Ознакомиться с основными способами создания и методами обработки строк в Java. Изучить принципы работы с потоками ввода-вывода.

**Краткие теоретические сведения**

Строка – упорядоченная последовательность символов. В Java строка является основным носителем информации. Для работы со строками используются классы: `String`, `StringBuilder` и `StringBuffer`, которые определены в пакете `java.lang` и доступны автоматически без объявления импорта.

Объекты класса `String` являются неизменяемыми(immutable)!. При попытке изменить строку создаётся новая строка. Все объекты класса `String` виртуальная машина хранит в пуле строк. Классы `StringBuffer` и `StringBuilder` допускают изменения в строке.

Класс **`String`**

***Создание строк***

1) Создание объекта класса `String`

```
String str1 = "Строка";
String str1 = new String("Строка");
```

****Примечание****

*Когда создается строка с использованием двойных кавычек, виртуальная машина ищет в пуле строк другую строку с таким же значением. Можно использовать метод `intern()` для сохранения строки в пуле строк, или получения ссылки, если такая строка уже находится в пуле.*

2) Получение строки из массива символов или байтов

```
char chars[] = { 'С', 'т', 'р', 'о', 'к', 'а' }:
String s = new String(chars);
```

3) Использование `StringBuffer и StringBuilder`

```
String(StringBuffer объект_StrBuf);
String(StringBuilder объект_StrBuild);
```

***Конкатенация (объединение) строк***

Оператор «+» - перегруженный оператор для класса `String` – используется для объединения строк.

```
String s1=”Hello”;
String s2=”World”;
s1+=”, ” + s2 + ” !”;
//s1 ссылается на строку “Hello, World !”
```

Если один из операндов в выражении содержит строку, то другие операнды также должны быть строками. Поэтому Java сама может привести переменные к строковому представлению, даже если они не являются строками.

***Разбиение строки на подстроки***

Метод `split(String regExp)` – разбивает строку на фрагменты, используя в качестве разделителей параметр `regExp`, и возвращает ссылку на массив, составленный из этих фрагментов. Сами разделители ни в одну подстроку не входят!

```
String parts[] = s1.split(“ ”);
//массив parts содержит {“Hello,”,”World”,”!”}
```

***Получение подстроки***

Метод `substring(int begin, int end)` – возвращает фрагмент исходной строки от символа с индексом `begin` до символа с индексом `end-1` включительно.

***Сравнение строк***

Метод `equals(Object obj)` – проверяет строки на полное совпадение. Возвращает `true` или `false`.

```
s1.equals(s2); //вернет false
```

Метод `equalsIgnoreCase(Object obj)` работает аналогично, но без учета регистра символов

****Примечание****

*Если сравнивать строки, используя логическую операцию “==”, то ее результатом будет* `true` *только в том случае, ели обе строковые переменные указывают (ссылаются) на один и тот же объект в памяти.*

***Замена отдельного символа***

Метод `replace(int old, int new)` возвращает новую строку, в которой все вхождения символа `old` заменены на символ `new`.

```
s1=s1.replace(‘и’, ‘э’);// s1=”Прэвет, Мэр!”
```

***Поиск подстроки***

Метод `indexOf(int ch)` возвращает индекс первого вхождения символа `ch` в исходную строку. Если применить метод в форме `indexOf(int ch, int i)`, то поиск вхождения начнется с символа с индексом `i`.

Последнее вхождение символа можно найти с помощью метода `lastIndexOf(int ch)`.

***Преобразование***

1) Число в строку

```
int integerVariable=10;
String intString=String.ValueOf(integerVariable);
```

2) Строка в число

```
String str="10";
int number=Integer.parseInt(str);
```

С остальными методами класса `String` ознакомьтесь самостоятельно!!!

Класс **`StringBuffer`**

Т.к. строки являются неизменными, их частая модификация методами класса `String` приводит к созданию новых объектов, что в свою очередь расходует память.

Для решения этой проблемы был создан класс `StringBuffer`, который позволяет более эффективно работать над модификацией строки. Класс `StringBuffer` может быть использован в многопоточных средах, т.к. все необходимые методы являются синхронизированными.

***Создание объекта* `StringBuffer`**

Каждый объект имеет свою вместимость (`capacity`),что отвечает за длину внутреннего буфера. Если длина строки, хранящейся в этом буфере, не превышает capacity, то нет необходимости выделять новый массив буфера. Если же буфер переполняется – он автоматически становится больше.

```
StringBuffer firstBuffer = new StringBuffer(); //capacity=16
StringBuffer firstBuffer = new StringBuffer("Hello"); //capacity=5+16
StringBuffer firstBuffer = new StringBuffer(50);//передаем capacity
```

***Модификация***

Используется для многократного выполнения операций добавления, вставки, удаления подстрок.

```
String domain=".com";
StringBuffer buffer=new StringBuffer("google");
buffer.append(domain); //"google.com"
buffer.delete(buffer.length()-domain.length()); //"google"
buffer.insert(buffer.length(), domain); //"google.com"
```

Класс **`StringBuilder`**

Данный класс представляет собой изменяемую последовательность символов. API идентичен `StringBuffer`. Единственное отличие – `StringBuilder` не синхронизирован. Достоинство – работает гораздо быстрее, чем `StringBuffer`.

В Java система ввода/вывода определяет концепцию потока данных – `Stream`. Введение концепции `Stream` позволяет отделить основную логику программы, обменивающейся информацией с любыми устройствами одинаковым образом, от низкоуровневых операций с такими устройствами ввода/вывода.

Можно выделить следующие общие для всех потоков этапы:

1) Открытие потока, связанного с определенным устройством: файл, URL и д.р.
2) Чтение/запись данных;
3) Закрытие потока.

Система ввода/вывода в Java представлена пакетами `java.io` и `java.nio`. Дополнительные классы потоков данных включены в пакеты `java.util.zip` и `java.util.jar`. Данные пакеты позволяют осуществлять работу с файлами и каталогами, читать и записывать информацию в байтовые и символьные потоки, архивировать/разархивировать и сериализовать объекты.

Обработка исключительных ситуаций производится при помощи следующих классов:

- **IOException** – корень иерархии исключений ввода-вывода. Вырабатывается всеми операциями ввода/вывода;
- **EOFException** – достигнут конец потока;
- **FileNotFoundException** – файл не найден;
- **UnsupportedEncodingException** – неизвестная кодировка.

***Работа с файлами и каталогами***

Для работы с физическим файлами и каталогами (директориями), расположенными на внешних носителях, в приложениях Java используются классы из пакета `java.io`.

Класс File служит для хранения и обработки в качестве объектов каталогов и имен файлов. Этот класс не содержит методы для работы с содержимым файла, но позволяет манипулировать такими свойствами файла, как права доступа, дата и время создания, путь в иерархии каталогов, создание, удаление файла, изменение его имени и каталога и т.д.

Объект класса `File` создается одним из нижеприведенных способов:

```
File myFile = new File(”\\com\\myfile.txt”);
File myDir  = new File(”c:\\jdk1.6.0\\src\\java\\io”);
File myFile = new File(myDir, ”File.java”);
File myFile = new File(”c:\\com”, ”myfile.txt”);
File myFile = new File(new URI(”Интернет-адрес”));
```

В первом случае создается объект, соответствующий файлу, во втором – подкаталогу. Третий и четвертый случаи идентичны. Для создания объекта указывается каталог и имя файла. В пятом – создается объект, соответствующий адресу в Интернете.

***Операции с файлами и каталогами***

- Проверка существования.
- Права доступа.
- Просмотр/поиск файлов и каталогов.
- Создание новых файлов и каталогов.
- Переименование файлов и каталогов.
- Удаление файлов и каталогов.

Класс `File`

При создании объекта класса File любым из конструкторов компилятор не выполняет проверку на существование физического файла с заданным путем.

Существует разница между разделителями, употребляющимися при записи пути к файлу: для системы Unix – “/”, а для Windows – “\\”. Для случаев, когда неизвестно, в какой системе будет выполняться код, предусмотрены специальные поля в классе File***:***

```
public static final String separator;
public static final char separatorChar;
```

С помощью этих полей можно задать путь, универсальный в любой системе:

```
File myFile = new File(File.separator+”com”
+ File.separator + ”myfile.txt” );
```

Также предусмотрен еще один тип разделителей – для директорий:

```
public static final String pathSeparator;
public static final char pathSeparatorChar;
```

***Проверка типа***

`isFile()` – является ли файлом

`isDirectory()` – является ли каталогом

`isHidden()` – является ли скрытым

***Получение информации о файле***

`exist()` – проверка существования

`length()` – длина файла

`lastModifier()` – время последней модификации

***Манипулирование правами***

- `boolean setReadable(boolean readable)`
- `boolean setReadable(boolean readable, boolean owner)`
- `boolean setWritable(boolean writable)`
- `boolean setWritable(boolean writable, boolean owner)`
- `boolean setExecutable(boolean executable)`
- `boolean setExecutable(boolean executable, boolean owner)`

***Получение информации о правах***

- `boolean canWrite()`
- `boolean canRead()`
- `boolean canExecute()`

***Просмотр файлов и каталогов***

- `String[] list()`
- `String[] list(FilenameFilter filter)`
- `File[] listFiles()`
- `File[] listFiles(FilenameFilter filter)`
- `File[] listFiles(FileFilter filter)`

***Просмотр файлов и каталогов***

– `boolean createNewFile() throws IOException` – создает новый пустой файл и возвращает `true`, если файл не существует, в противном случае возвращается `false`.

– `boolean mkdir()`

– `boolean mkdirs()` – создает промежуточные родительские директории.

***Переименование файлов и каталогов***

`boolean renameTo(File dest)` – переименовывает файл/каталог, генерируется SecurityException при отказе в доступе.

***Удаление файлов и каталогов***

`boolean delete()` – удаляет файл/каталог, каталог должен быть пустым, генерируется `SecurityException` при отказе в доступе.

Пример работы с классом `File`.

```
File fp = new File("chapt09" + File.separator + "FileTest2.java");

if (fp.exists()) {

  if (fp.isFile()) {

  System.out.println("Путь к файлу:\t" + fp.getPath());

  System.out.println("Абсолютный путь:\t"+fp.getAbsolutePath());

  System.out.println("Размер файла:\t" + fp.length());

  System.out.println("Последняя модификация:\t"+new Date(fp.lastModified()));

  System.out.println("Файл доступен для чтения:\t"+ fp.canRead());

  System.out.println("Файл доступен для записи:\t"+ fp.canWrite());

  System.out.println("Файл удален:\t" + fp.delete());    }

  } else
```

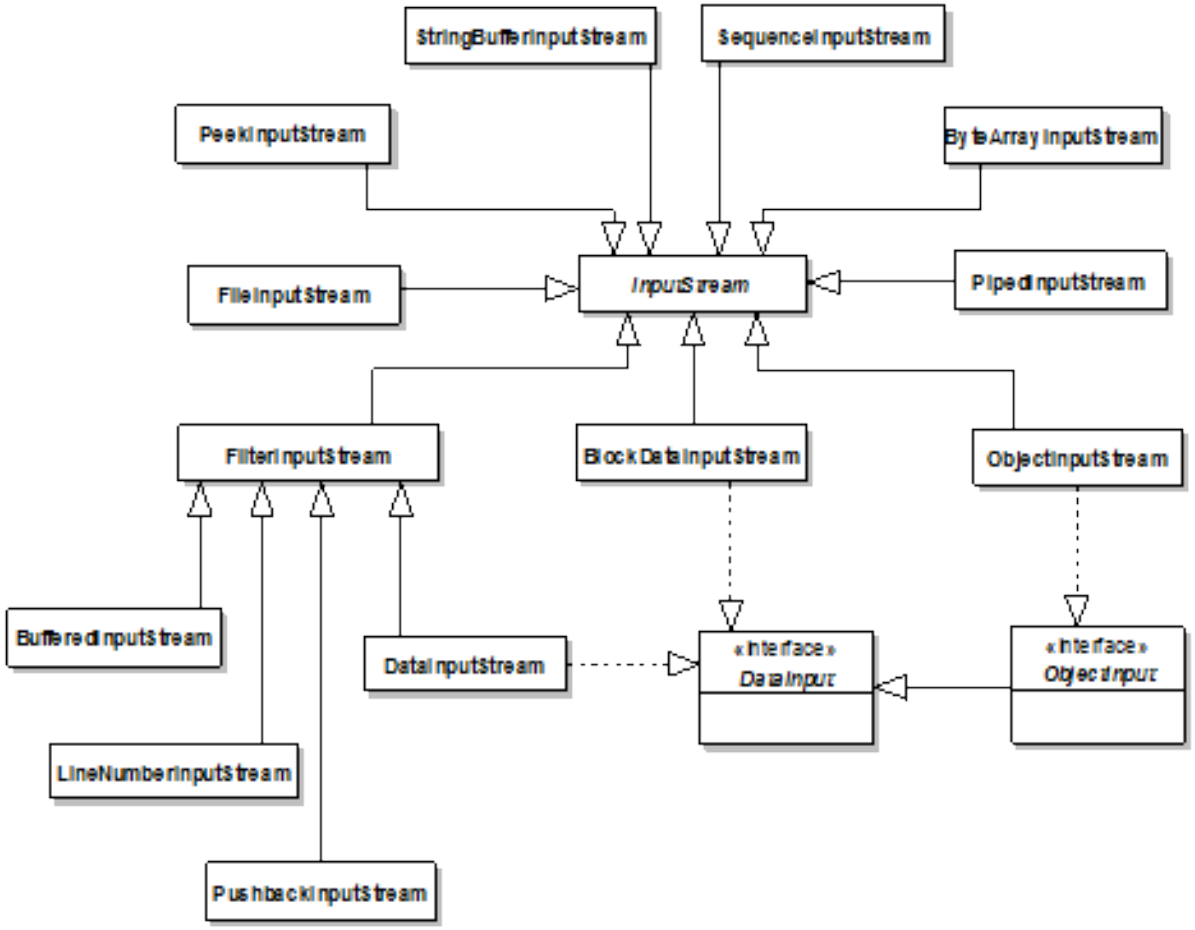
```
    System.out.println("файл"+fp.getName()+"не существует");
    try {
    if (fp.createNewFile())
    System.out.println("Файл"+fp.getName()+"создан");
    } catch (IOException e) {
    System.err.println(e); }

File dir = new File("com" + File.separator + "learn");
if (dir.exists() && dir.isDirectory()){
    System.out.println("каталог "+ dir.getName() + " существует");
    File[] files = dir.listFiles();
    for (int i = 0; i < files.length; i++) {
        Date date = new Date(files[i].lastModified());
        System.out.print("\n" + files[i].getPath() + " \t| "  +
files[i].length() + "\t| " + date.toString())}
    File root = File.listRoots()[1];
    System.out.printf("\n%s %,d из %,d свободно.", root.getPath(),
root.getUsableSpace(), root.getTotalSpace());}
```

***Байтовые потоки***

Потоки ввода последовательности байтов являются подклассами абстрактного класса `InputStream`, потоки вывода – подклассами абстрактного класса `OutputStream`. Эти классы являются суперклассами для ввода массивов байтов, строк, объектов, а также для выбора из файлов и сетевых соединений. При работе с файлами используются подклассы этих классов соответственно `FileInputStream` и `FileOutputStream`, конструкторы которых открывают поток и связывают его с соответствующим физическим файлом.

Поток ввода связывается с одним из источников данных, в качестве которых могут быть использованы массив байтов, строка, файл, «pipe»-канал, сетевые соединения и др. Набор классов для взаимодействия с перечисленными источниками приведен на рисунке.




`InputStream` – это базовый класс для потоков ввода. Простейшая операция представлена абстрактным методом `read()`. Этот метод предназначен для считывания ровно одного байта из потока, однако возвращает при этом значение типа int. Если считывание произошло успешно, возвращаемое значение лежит в диапазоне от 0 до 255. Если достигнут конец потока, то возвращаемое значение равно -1. Если же считать из потока данные не удается из-за каких-то ошибок, будет сгенерировано исключение `java.io.IOException`.

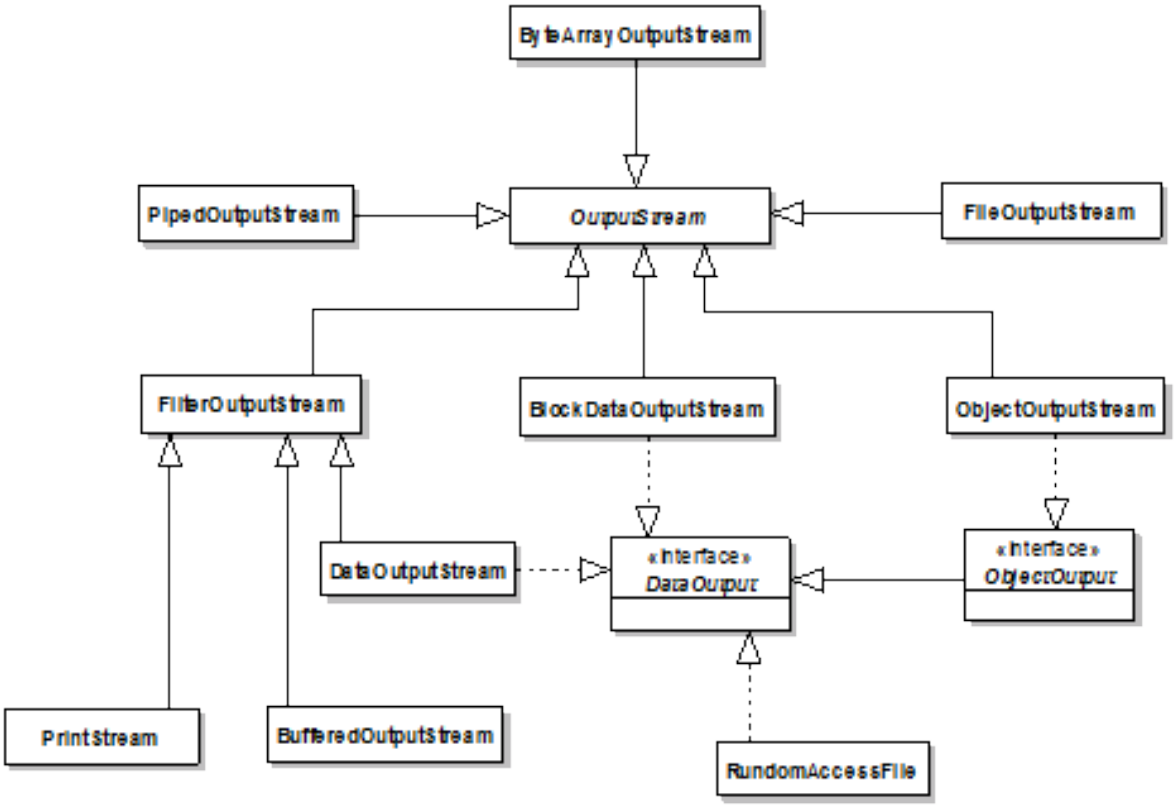
Для считывания нескольких байт подряд используется метод `read()`, где в качестве параметров передается массив `byte[]`. Если необходимо заполнить не весь массив, а только его часть, то для этих целей используется метод `read()`, которому, кроме массива `byte[]`, передаются два `int` значения: позиция в массиве, с которой следует начать заполнение и количество байт, которое нужно считать. При вызове методов `read()` возможно возникновение такой ситуации, когда запрашиваемые данные еще не готовы к считыванию. В таком случае выполнение останавливается на вызове метода `read()` и вызывающий поток блокируется.

Чтобы узнать, сколько байт в потоке готово к считыванию, применяется метод `available()`. Метод `available()` возвращает число – количество байт, именно на данный момент готовых к считыванию.

Когда работа с входным потоком данных окончена, его следует закрыть. Для этого вызывается метод `close()`. Этим вызовом будут освобождены все системные ресурсы, связанные с потоком.

Абстрактный класс `FilterInputStream` используется как шаблон для настройки классов ввода, наследуемых от класса `InputStream`. Класс `DataInputStream` предоставляет методы для чтения из потока данных значений базовых типов, но начиная с версии 1.2 класс был помечен как `deprecated` и не рекомендуется к использованию. Класс `BufferedInputStream` присоединяет к потоку буфер для ускорения последующего доступа.

Для вывода данных используются потоки следующих классов.



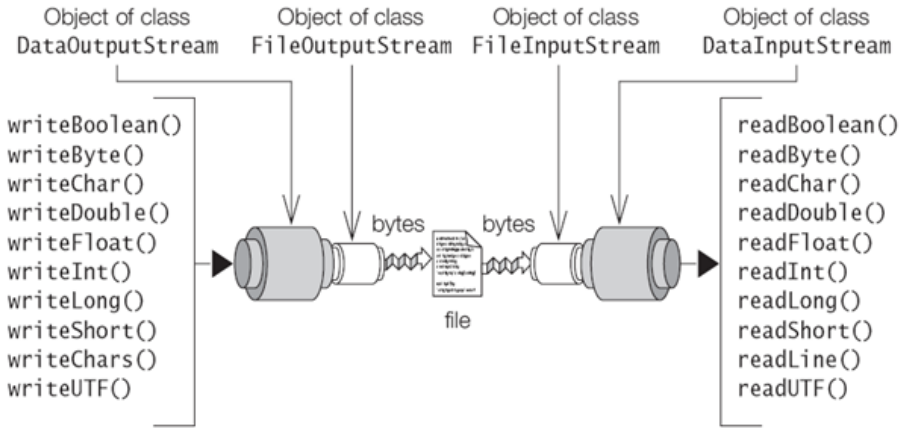

Класс `OutputStream` – это базовый класс для потоков вывода, в нем аналогичным образом определяются три метода `write()`.

В случае ошибки могут быть сгенерированы исключения `java.io.IOException` или `IndexOutOfBoundsException`.

Класс выходного потока может использовать внутренний механизм для буферизации данных. Чтобы убедиться, что данные записаны в поток, а не хранятся в буфере, вызывается метод `flush()`, определенный в `OutputStream`.

Когда работа с потоком закончена, его следует закрыть. Для этого вызывается метод `close()`. Этот метод сначала освобождает буфер (вызовом метода `flush()`), после чего поток закрывается и освобождаются все связанные с ним системные ресурсы.

Схема чтения/записи бинарных файлов




Для чтения бинарного представления значений примитивных типов Java необходимо:

1. Создать `FileInputStream`:

```
FileInputStream inputFile = new FileInputStream("primitives.data");
```

2. Создать `DataInputStream`, который связан с `FileInputStream`:

```
DataInputStream inputStream = new DataInputStream(inputFile);
```

3. Считать (точное количество) примитивных значений `Java` в том же порядке, в котором они были записаны, используя методы `readX()`:

```
boolean v = inputStream.readBoolean();

char c = inputStream.readChar();

byte b = inputStream.readByte();

short s = inputStream.readShort();

int i = inputStream.readInt();

long l = inputStream.readLong();

float f = inputStream.readFloat();

double d = inputStream.readDouble();
```

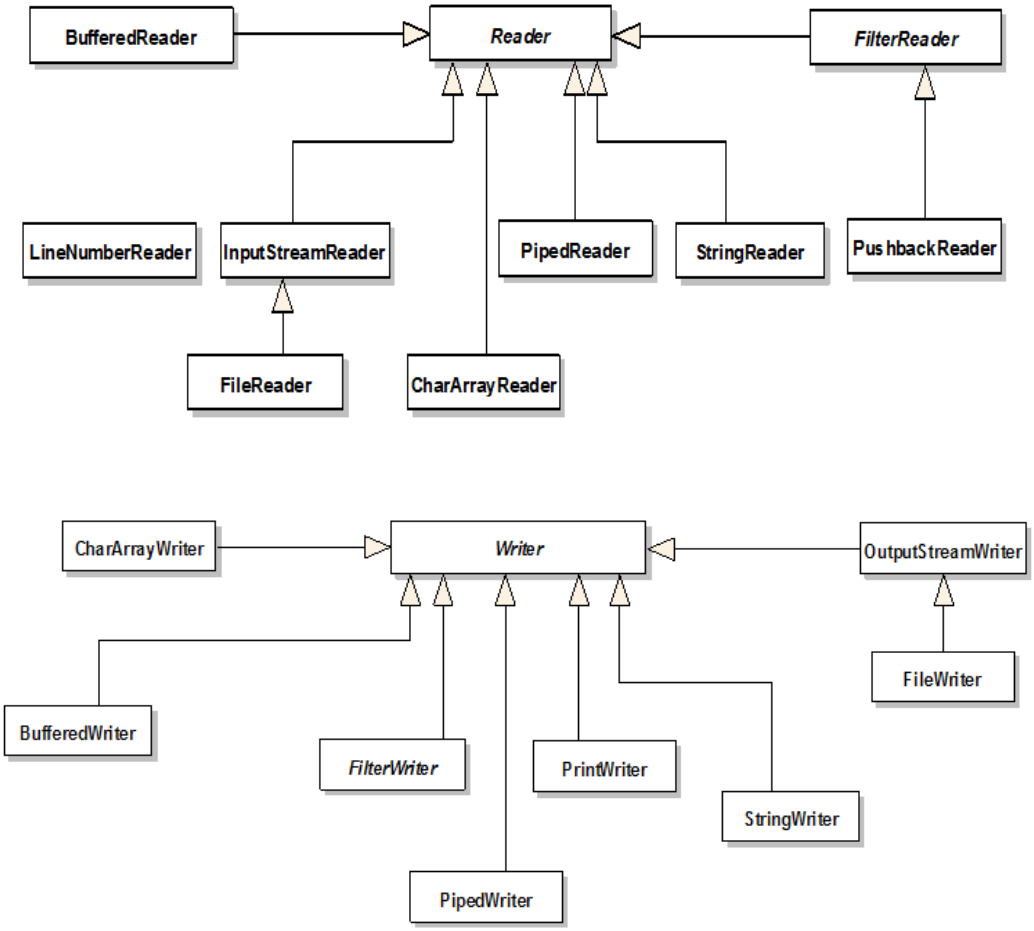
4. Закрыть `поток inputStream`, что вызовет закрытие связанных потоков

```
inputStream.close();
```

### *Символьные потоки*

Для обработки символьных потоков в формате Unicode применяется отдельная иерархия подклассов абстрактных классов Reader и Writer, которые почти полностью повторяют функциональность байтовых потоков, но являются более актуальными при передаче текстовой информации.

Например, аналогом класса FileInputStream является класс FileReader. Такой широкий выбор потоков позволяет выбрать наилучший способ записи в каждом конкретном случае.




```
//пример: чтение по одному байту (символу) из потока ввода
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
public class ReadDemo {
public static void main(String[] args) {
File f = new File("file.txt");//должен существовать!
int b, count = 0;
try {
FileReader is = new FileReader(f);
```

```
/* FileInputStream is = new FileInputStream(f);*/ / альтернатива
while ((b = is.read()) != -1) {/*чтение*/
System.out.print((char)b);
count++; }

is.close(); // закрытие потока ввода
} catch (IOException e) {
System.err.println("ошибка файла: " + e); }
System.out.print("\n число байт = " + count); }}
```

Один из конструкторов `FileReader(f)` или `FileInputStream(f)` открывает поток `is` и связывает его с файлом `f`. Для закрытия потока используется метод `close()`. При чтении из потока можно пропустить `n` байт с помощью метода `long skip(long n)`.

Для вывода символа (байта) или массива символов (байтов) в поток используются потоки вывода – объекты подкласса `FileWriter` суперкласса `Writer` или подкласса `FileOutputStream` суперкласса `OutputStream`. В следующем примере для вывода в связанный с файлом поток используется метод `write()`.

```
// пример: вывод массива в поток в виде символов и байтов:
import java.io.*;
public class WriteRunner {
public static void main(String[] args) {
String pArray[] = { "2007 ", "Java SE 6" };

File fbyte = new File("byte.txt");
File fsymb = new File("symbol.txt");
try {
FileOutputStream fos = new FileOutputStream(fbyte);

FileWriter fw = new FileWriter(fsymb);
for (String a : pArray) {
fos.write(a.getBytes());

fw.write(a);}

fos.close();

fw.close();

} catch (IOException e) {
System.err.println("ошибка файла: " + e);}}}
```

В результате будут получены два файла с идентичным набором данных, но созданные различными способами.

В отличие от классов `FileInputStream` и `FileOutputStream` класс `RandomAccessFile` позволяет осуществлять произвольный доступ к потокам как ввода, так и вывода. Поток рассматривается при этом как массив байтов, доступ к элементам осуществляется с помощью метода `seek(long poz)`. Для создания потока можно использовать один из конструкторов:

```
RandomAccessFile(String name, String mode);
RandomAccessFile(File file, String mode);
Параметр mode равен "r" для чтения или "rw" для чтения и записи.

/* пример: запись и чтение из потока: */
import java.io.*;
public class RandomFiles {
public static void main(String[] args) {
double data[] = { 1, 10, 50, 200, 5000 };
try {
RandomAccessFile rf = new RandomAccessFile("temp.txt", "rw");

for (double d : data)
rf.writeDouble(d); // запись в файл
/* чтение в обратном порядке */
for (int i = data.length - 1; i >= 0; i--) {
rf.seek(i * 8);

// длина каждой переменной типа double равна 8-и байтам
System.out.println(rf.readDouble());
}

rf.close();
 } catch (IOException e) {

System.err.println(e);}}}
```

### *Работа с zip-архивами*

Кроме общего функционала для работы с файлами Java предоставляет функциональность для работы с таким видом файлов как zip-архивы. Для этого в пакете `java.util.zip` определены два класса – `ZipInputStream` и `ZipOutputStream`

*Чтение архивов*

Для чтения архивов применяется класс `ZipInputStream`. В конструкторе он принимает поток, указывающий на `zip`-архив:

```
ZipInputStream(InputStream in)
```

Для считывания файлов из архива `ZipInputStream` использует метод `getNextEntry()`, который возвращает объект `ZipEntry`. Объект `ZipEntry` представляет отдельную запись в `zip`-архиве.

```
public class FilesApp {
 public static void main(String[] args) {
 try (ZipInputStream zin = new ZipInputStream (new FileInputStream("C:\SomeDir\notes3.zip")))
 {
 ZipEntry entry;
 String name;
  long size;
 while((entry=zin.getNextEntry())!=null){
 name = entry.getName(); // получим название файла
 size=entry.getSize();  // получим его размер в байтах
 System.out.printf("Название: %s \t размер: %d \n", name, size);
 }
 }catch(Exception ex){
    System.out.println(ex.getMessage());
 } } }
```

`ZipInputStream` в конструкторе получает ссылку на поток ввода. И затем в цикле выводятся все файлы и их размер в байтах, которые находятся в данном архиве.

*Запись архивов*

Для создания архива используется класс `ZipOutputStream`. Для создания объекта `ZipOutputStream` в его конструктор передается поток вывода:

```
ZipOutputStream(OutputStream out)
```

Для записи файлов в архив для каждого файла создается объект `ZipEntry`, в конструктор которого передается имя архивируемого файла. А чтобы добавить каждый объект `ZipEntry` в архив, применяется метод `putNextEntry()`.

```
import java.io.*;
import java.util.zip.*;

public class FilesApp {
```

```
      public static void main(String[] args) {
          String filename = "C:\SomeDir\notes3.txt";
          try(ZipOutputStream zout = new ZipOutputStream(new
FileOutputStream("C:\SomeDir\output.zip"));
                  FileInputStream fis= new
FileInputStream(filename);)
          {
              ZipEntry entry1=new ZipEntry(filename);
              zout.putNextEntry(entry1);
              // считываем содержимое файла в массив byte
              byte[] buffer = new byte[fis.available()];
              fis.read(buffer);
              // добавляем содержимое к архиву
              zout.write(buffer);
              // закрываем текущую запись для новой записи
              zout.closeEntry();
          }
          catch(Exception ex){

              System.out.println(ex.getMessage());
          }}}
```

После добавления объекта `ZipEntry` в поток надо добавить в него и содержимое файла. Для этого используется метод write, записывающий в поток массив байтов: `zout.write(buffer);`. В конце надо закрыть `ZipEntry` с помощью метода closeEntry(). После этого можно добавлять в архив новые файлы - в этом случае все вышеописанные действия для каждого нового файла повторяются.

**Задание:**

1. Изучите методы для работы со строками.
2. Напишите метод проверки, является ли строка палиндромом (Строка считается полиндромом, если она одинаково читается в обоих направлениях)
3. Замените в строке все вхождения одного слова на другое.
4. ***Напишите программу, печатающую все перестановки строки (из 4 символов)
5. Изучите классы и методы для работы с файлами.
6. Программе на вход подается имя каталога. Необходимо найти все текстовые файлы в этом каталоге, считать их содержимое и записать в новый файл. Полученный файл нужно упаковать в zip-архив.