

# Система ввода/вывода

Потоки данных (stream)  
Сериализация объектов



# Система ввода-вывода

- Подавляющее большинство программ обменивается данными с внешним миром.
- Устройства, откуда может производиться считывание информации, могут быть самыми разнообразными - файл, клавиатура, (входящее) сетевое соединение и т.п.
- То же самое касается и устройств вывода - это может быть файл, экран монитора, принтер, (исходящее) сетевое соединение и т.п.
- В Java система ввода-вывода представлена в `java.io.*`

# Поток (stream)

- Для описания работ по вводу/выводу используется специальное понятие – поток
- Поток данных связан с некоторым **источником** или **приемником**, способным получать или передавать данные. Соответственно, потоки делятся на **входящие** – читающие данные, и **исходящие** – записывающие данные.
- Потоки представляются **объектами**, описанными в java.io. Они довольно разные и отвечают за различную функциональность.

# Поток (stream)

- *Поток* является абстракцией, которая или производит, или потребляет информацию.
- Поток связывается с физическим устройством с помощью системы ввода/вывода Java (Java I/O system).
- Все потоки ведут себя одинаковым образом, хотя фактические физические устройства, с которыми они связаны, могут сильно различаться.
- Таким образом, одни и те же классы и методы ввода/вывода можно применять к устройствам любого типа.
- Благодаря потокам ваша программа выполняет ввод/вывод, не понимая различий между клавиатурой и сетью.
- Java реализует потоки с помощью иерархии классов, определенных в пакете `java.io`.

# Считывание и запись информации

- Существующие стандартные классы помогают решить большинство типичных задач.
- Минимальной "порцией" информации является бит, принимающий значение 0 или 1. Традиционно используется более крупная единица измерения байт, объединяющая 8 бит.
- Базовые, наиболее универсальные классы позволяют считывать и записывать информацию именно в виде набора байт. Чтобы их было удобно применять в различных задачах, `java.io` содержит также классы, преобразующие любые данные в набор байт.
- Понятно, что при восстановлении данных выполняются обратные действия - сначала считывается последовательность байт, а затем она преобразовывается в нужный формат.

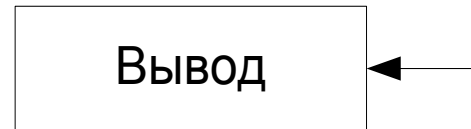
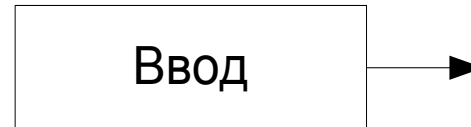
# Байтовые и символьные потоки

- Java 2 определяет два типа потоков – байтовый и символьный.
- *Байтовые потоки* предоставляют удобные средства для обработки ввода и вывода байтов. Байтовые потоки используются, например, при чтении или записи данных в двоичном коде.
- *Символьные потоки* предоставляют удобные средства для обработки ввода и вывода символов. Они используют Unicode и поэтому могут быть интернационализированы. Кроме того, в некоторых случаях символьные потоки более эффективны, чем байтовые.
- На самом низком уровне весь ввод/вывод все еще байтовый. Символьно-ориентированные потоки обеспечивают удобные и эффективные средства для обработки символов.

# Виды потоков

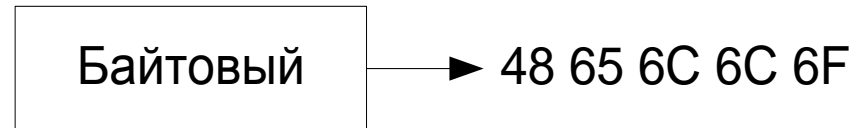
- Направление

- Ввод
- Вывод



- Содержимое

- Байтовые
- Символьные



# Классы потоков

	Байтовый	Символьный
Ввод	InputStream	Reader
Вывод	OutputStream	Writer

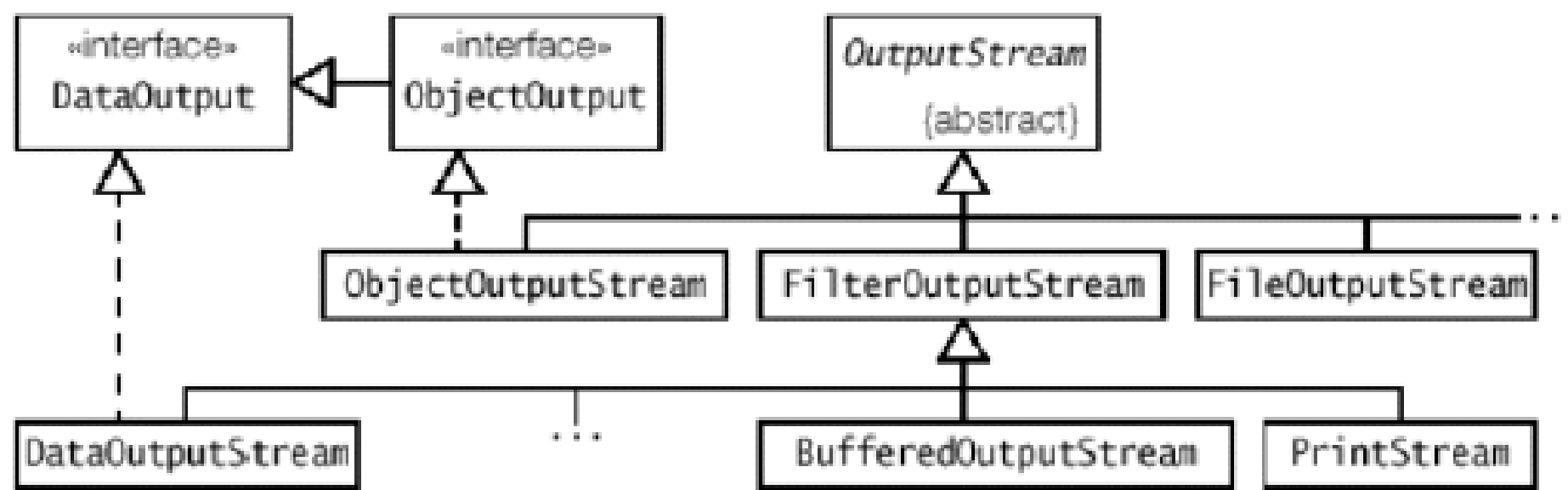
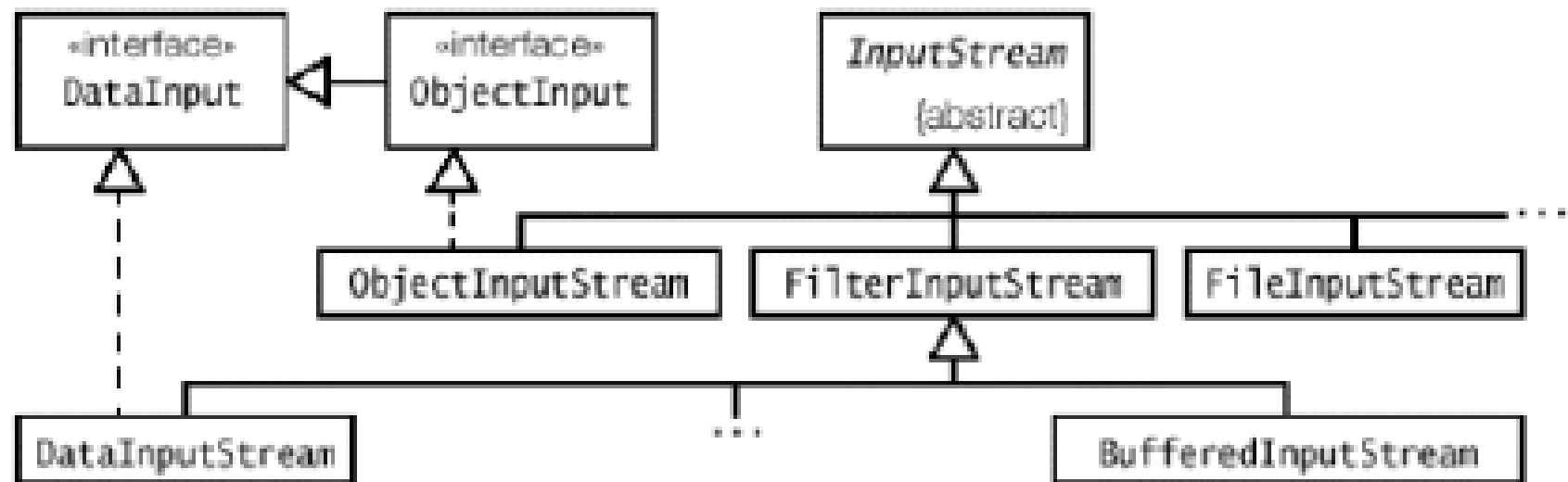


# Исключительные ситуации

- Класс `IOException`
  - Корень иерархии исключений ввода-вывода
  - Бросается всеми операциями ввода/вывода
- Класс `EOFException`
  - Достигнут конец потока
- Класс `FileNotFoundException`
  - Файл не найден
- Класс `UnsupportedEncodingException`
  - Неизвестная кодировка

# Классы байтовых потоков

- Байтовые потоки определяются в двух иерархиях классов.
- Наверху этой иерархии — два абстрактных класса: `InputStream` и `OutputStream`.
- Каждый из этих абстрактных классов имеет несколько конкретных подклассов, которые обрабатывают различия между разными устройствами, такими как дисковые файлы, сетевые соединения и даже буферы памяти.
- Абстрактные классы `InputStream` и `OutputStream` определяют несколько ключевых методов, которые реализуются другими поточными классами. Два наиболее важных — `read()` и `write()`, которые, соответственно, читают и записывают байты данных.
- Оба метода объявлены как абстрактные внутри классов `InputStream` и `OutputStream` и переопределяются производными поточными классами.



# Классы байтовых потоков

Поточный класс	Значение
BufferedInputStream	Буферизованный поток ввода
BufferedOutputStream	Буферизованный поток вывода
ByteArrayInputStream	Поток ввода, который читает из байт-массива
ByteArrayOutputStream	Поток вывода, который записывает в байт-массив
DataInputStream	Поток ввода, который содержит методы для чтения данных стандартных типов
DataOutputStream	Поток вывода, который содержит методы для записи данных стандартных типов
FileInputStream	Поток ввода, который читает из файла
FileOutputStream	Поток вывода, который записывает в файл
FilterInputStream	Реализует InputStream
FilterOutputStream	Реализует OutputStream
InputStream	Абстрактный класс, который описывает поточный ввод
OutputStream	Абстрактный класс, который описывает поточный вывод
PipedInputStream	Канал ввода
PipedOutputStream	Канал вывода
Printstream	Поток вывода, который поддерживает print() и println()
PushbackInputStream	Поток (ввода), который поддерживает однобайтовую операцию "unget", возвращающую байт в поток ввода
RandomAccessFile	Поддерживает ввод/вывод файла произвольного
SequenceInputStream	Поток ввода, который является комбинацией двух или нескольких потоков

# Базовый класс ввода – InputStream

- Это базовый класс для потоков ввода, т.е. чтения. Он описывает базовые методы для работы с байтовыми потоками данных.
- Метод ***int read()*** – абстрактный. Считывает один байт из потока. Если чтение успешно, то значение между 0..255 представляет собой полученный байт, если конец потока, то значение равно -1.
- Метод ***int read(byte[] b)*** возвращает количество считанных байт и сохраняет их в массив ***b***. Метод так же абстрактный.

# Класс InputStream

- Метод ***read(byte[] b, int offset, int length)*** считывает в массив ***b*** начиная с позиции ***offset*** ***length*** символов.
- Метод ***int available()*** возвращает количество байт, на данный момент готовых к чтению из потока. Этот метод применяют, что бы операции чтения не приводили к зависанию, когда данные не готовы к считыванию.
- Метод ***close()*** заканчивает работу с потоком.

# Класс OutputStream

- Базовый класс для работы с выводом. Его функции аналогичны функциям предыдущего класса, но метод `read()` заменяется на `write()`.
- Абстрактные методы записи:
  - **`void write(int i)`** (пишет только старшие 8 бит, остальные игнорирует)
  - **`void write(byte[] b)`**
  - **`void write(byte[] b, int offset, int length)`**
- Метод **`flush()`** для очистки буфера и записи данных
- Метод **`close()`** для закрытия потока

# Классы ByteArrayInputStream

- Поток, считывающий (записывающий) данные из массива байт. У класса **ByteArrayInputStream** конструктор принимает массив **byte[]**. Тогда при вызове **read()** данные будут браться из него.

```
byte[] bytes = {1, -1, 0};  
ByteArrayInputStream in = new ByteArrayInputStream(bytes);  
int readedInt = in.read(); //readedInt=1  
readedInt = in.read();     //readedInt=255  
readedInt = in.read();     //readedInt=0
```



# Класс ByteArrayOutputStream

- Применяется для записи в массив. Этот класс внутри себя использует объект ***byte[]*** для хранения данных. Получить данный объект можно с помощью ***toByteArray()***

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

# Чтение и запись файлов

- Для Java все файлы имеют байтовую структуру, а Java обеспечивает методы для чтения и записи байтов в файл.
- Java позволяет упаковывать *байтовый* файловый поток в символично-ориентированный объект.
- Для создания байтовых потоков, связанных с файлами, чаще всего используются два поточных класса — `FileInputStream` и `FileOutputStream`.

**`FileInputStream (String fileName)` throws `FileNotFoundException`**

**`FileOutputStream (String fileName)` throws `FileNotFoundException`**

где *fileName* определяет имя файла, который вы хотите открыть.

- Когда выходной файл открывается, любой файл, существовавший ранее с тем же самым именем, разрушается.

# Классы `FileInputStream` и `FileOutputStream`

- Классы используются для чтения и записи в файл соответственно. Для открытия файла на чтение в конструкторе класса указывается имя файла. При записи в файл кроме имени можно указывать и доступ.
- При вызове одного из трех методов **`write()`** данные будут переданы в файл. По окончании работы с этими классами необходимо вызвать **`close()`**. Применяется так же метод **`available()`** для считывания из файла.

```
byte[] bytesToWrite ={1,2,3};      //что записываем
byte[] bytesReaded =new byte[10]; //куда считываем
String fileName ="d:\\test.txt";

try {
    FileOutputStream outFile =new FileOutputStream(fileName);
    outFile.write(bytesToWrite);      //запись в файл
    outFile.close();

    FileInputStream inFile =new FileInputStream(fileName);
    int bytesAvailable =inFile.available(); //сколько можно считать
    int count =inFile.read(bytesReaded,0,bytesAvailable);
    inFile.close();}

catch (FileNotFoundException e){
    System.out.println("Невозможно произвести запись в файл:"+fileName);}
catch (IOException e){
    System.out.println("Ошибка ввода/вывода:"+e.toString());}
```

# Класс `SequenceInputStream`

- Объединяет поток данных из двух и более других входных потоков. Данные считываются последовательно – все из первого, затем второго..
- Для данного класса существует два конструктора. Если необходимо объединить два потока, то конструктор принимает через запятую их. Если более двух, то передается ***Enumeration*** с экземплярами класса ***InputStream***.

# Пример

```
FileInputStream inFile1 =null;
FileInputStream inFile2 =null;
SequenceInputStream sequenceStream =null;
FileOutputStream outFile =null;

inFile1 =new FileInputStream("file1.txt");
inFile2 =new FileInputStream("file2.txt");
sequenceStream =new SequenceInputStream(inFile1,inFile2);
outFile =new FileOutputStream("file3.txt");
int readedByte =sequenceStream.read();
    while(readedByte!=-1){outFile.write(readedByte);
readedByte =sequenceStream.read();
```

# Классы-фильтры (надстройки)

- Задачи, возникающие при *i/o* операциях разнообразны. Применяются надстройки – наложение дополнительных объектов для получения новых свойств и функций.
- Классы ***FilterInputStream*** и ***FilterOutputStream*** унаследованы от базовых ***InputStream*** и ***OutputStream***. Надстройки принимают в качестве аргумента конструктора экземпляры классов ***InputStream*** и ***OutputStream***. Надстройки не являются источником данных, они лишь модифицируют работу надстраиваемого потока.

# Класс-надстройка `BufferedInputStream` и `BufferedOutputStream`

- Чтение и запись некоторых данных происходит из буфера, что ускоряет процесс в несколько раз.
- Метод ***mark()*** запоминает точку во входном потоке, метод ***reset()*** приводит к тому, что все данные, полученные после ***mark()***, считываются повторно.
- Запись в устройство вывода произойдет, когда буфер заполнится, но можно это сделать явно, вызвав метод ***flush()*** класса ***BufferedOutputStream***.



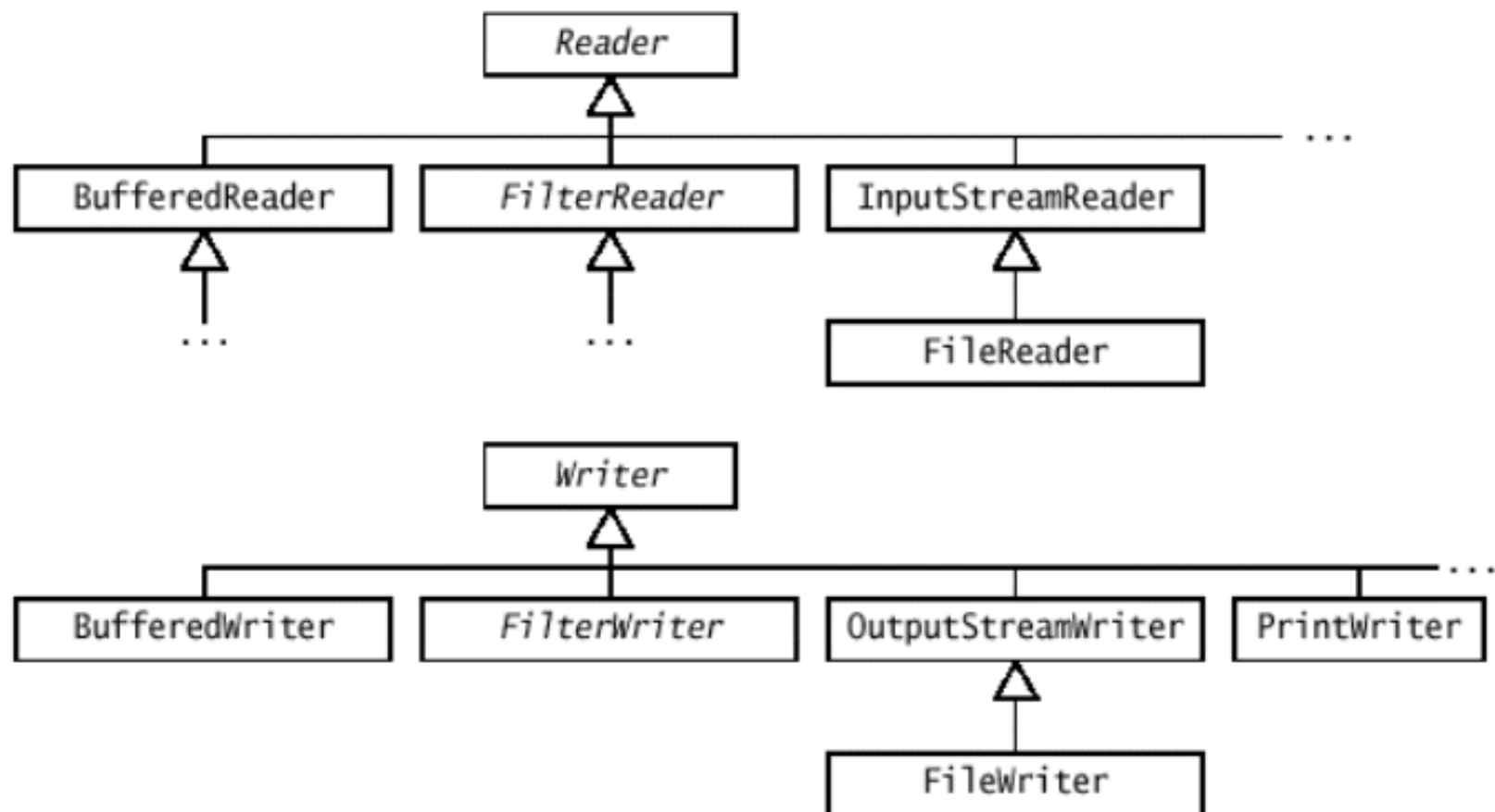
# Пример на запись и считывание с буферизацией

```
String fileName = "d:\\file1";
InputStream inStream = null;
OutputStream outStream = null;
//Записать в файл некоторое количество байт
long timeStart = System.currentTimeMillis();
outStream = new FileOutputStream(fileName);
outStream = new BufferedOutputStream(outStream);
for(int i=1000000;--i>=0;){
    outStream.write(i);}
inStream = new FileInputStream(fileName);
inStream = new BufferedInputStream(inStream);
while(inStream.read()!=-1){...
}
```

# Классы символьных потоков

- Символьные потоки определены в двух иерархиях классов. Наверху этой иерархии два абстрактных класса: `Reader` и `Writer`.
- Они обрабатывают потоки символов Unicode.
- Абстрактные классы `Reader` и `Writer` определяют несколько ключевых методов, которые реализуются другими поточными классами.
- Два самых важных метода — `read()` и `write()`, которые читают и записывают символы данных, соответственно.
- Они переопределяются производными поточными классами.

# СИМВОЛЬНЫЕ ПОТОКИ



# Классы символьных потоков

Поточный класс	Значение
BufferedReader	Буферизированный символьный поток ввода
BufferedWriter	Буферизированный символьный поток вывода
CharArrayReader	Поток ввода, который читает из символьного массива
CharArrayWrite	Выходной поток, который записывает в символьный массив
FileReader	Поток ввода, который читает из файла
FileWriter	Выходной поток, который записывает в файл
FilterReader	Отфильтрованный поток ввода
FilterWriter	Отфильтрованный поток вывода
InputStreamReader	Поток ввода, который переводит байты в символы
LineNumberReader	Поток ввода, который считает строки
OutputStreamWriter	Поток ввода, который переводит символы в байты
PipedReader	Канал ввода
PipedWriter	Канал вывода
PrintWriter	Поток вывода, который поддерживает print() и println()
PushbackReader	Поток ввода, возвращающий символы в поток ввода
Reader	Абстрактный класс, который описывает символьный поток ввода
StringReader	Поток ввода, который читает из строки
StringWriter	Поток вывода, который записывает в строку
Writer	Абстрактный класс, который описывает символьный поток вывода

# Надстройки DataInputStream и DataOutputStream

- До сих пор речь шла о считывании и записи в поток только типа **byte**. Для работы с другими примитивными типами существуют интерфейсы **DataInput** и **DataOutput** и их реализация в виде классов.
- Запись производится методом **writeXxx()**, где Xxx – тип данных – Int, Long, Double, Byte.
- Для считывания необходимо соблюдать и применять те операторы **readXxx**, в котором были записаны типы данных с помощью **writeXxx()**.

# Пример на запись и считывание СИМВОЛЬНОГО ПОТОКА

```
String fileName = "d:\\file.txt";
FileWriter fw = null; BufferedWriter bw = null;
FileReader fr = null; BufferedReader br = null;
//Строка, которая будет записана в файл
String data = "Some data to be written and readed \n";
try{
fw = new FileWriter(fileName); bw = new BufferedWriter(fw);
System.out.println("Write some data to file: " + fileName);
// Несколько раз записать строку
for(int i=(int)(Math.random()*10);--i>=0;)bw.write(data);
bw.close();
fr = new FileReader(fileName); br = new BufferedReader(fr);
String s = null;
int count = 0;
System.out.println("Read data from file: " + fileName);
// Считывать данные, отображая на экран
while((s=br.readLine())!=null)
System.out.println("row " + ++count + " read:" + s);
br.close();
}catch(Exception e){ e.printStackTrace(); }
```

# Предопределенные потоки

- Все программы Java автоматически импортируют пакет `java.lang`.
- Этот пакет определяет класс с именем `System`, инкапсулирующий некоторые аспекты исполнительной среды Java.
- Класс `System` содержит также три предопределенные поточные переменные ***in***, ***out*** и ***err***. Эти поля объявлены в `System` со спецификаторами `public` и `static`.
- Объект **`System.out`** называют *поток стандартного вывода*. По умолчанию с ним связана консоль.
- На объект **`System.in`** ссылаются как на *стандартный ввод*, который по умолчанию связан с клавиатурой.
- К объекту **`System.err`** обращаются как к *стандартному потоку ошибок*, который по умолчанию также связан с консолью.
- Однако эти потоки могут быть переназначены на любое совместимое устройство ввода/вывода.

# Чтение консольного ввода

- Консольный ввод в Java выполняется с помощью считывания из объекта `System.in`.
- Чтобы получить символьный поток, который присоединен к консоли, вы переносите ("упаковываете") `System.in` в объект типа `BufferedReader`.

## **BufferedReader(Reader inputReader)**

- где **inputReader** — поток, который связан с создающимся экземпляром класса `BufferedReader`.
- `Reader` — абстрактный класс. Один из его конкретных подклассов — это `InputStreamReader`, который преобразовывает байты в символы.
- Чтобы получить `InputStreamReader`-объект, который связан с `System.in`, используйте следующий конструктор:

## **InputStreamReader(InputStream inputstream)**

- Поскольку `System.in` ссылается на объект типа `InputStream`, его можно использовать в качестве параметра **inputstream**.
- Следующая строка кода создает объект класса `BufferedReader`, который связан с клавиатурой:

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

- После того как этот оператор выполнится, объектная переменная `br` станет символьным потоком, связанным с консолью через `System.in`.



# Чтение символов

```
// Использует BufferedReader для чтения символов с консоли,
import java.io.*;
class BRRead
{
    public static void main(String args [ ])
        throws IOException
    {
        char c;
        BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
        System.out.println("Введите символы, 'q' - для завершения.");
        // чтение символов
        do
        {
            c = (char) br.read();
            System.out.println(c);
        }
        while(c != 'q');
    }
}
```

# Чтение строк

```
// Крошечный редактор.
import java.io.*;
class TinyEdit {
public static void main(String args[])
throws IOException {
// Создать BufferedReader-объект, используя System.in
BufferedReader br = new BufferedReader(new
    InputStreamReader(System.in));
String str[] = new String[100];
System.out.println("Введите строки текста.");
System.out.println("Введите 'stop' для завершения.");
for (int i=0; i<100; i++)
{
str[i] = br.readLine();
if(str[i].equals("stop"))
break; }
System.out.println("\n Вот ваш файл:");
// Вывести строки на экран.
for (int i=0; i<100; i++) {
if(str[i].equals("stop")) break;
System.out.println(str[i]); }
}}
```

# Запись консольного вывода

- Консольный вывод легче всего выполнить с помощью методов `print ()` и `println()`.
- Эти методы определены классом `Printstream` (который является типом (классом) объекта `System.out`).
- Поскольку `PrintStream` — выходной поток, производный от `OutputStream`, он также реализует метод нижнего уровня `write()`.
- Самая простая форма `write ()`, определенная в `Printstream`, имеет вид:

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

- Этот метод записывает в файл байт, указанный в параметре *byteval*. Хотя *byteval* объявлен как целое число, записываются только младшие восемь битов.

# Запись консольного вывода

```
// Демонстрирует System.out.write.  
class WriteDemo {  
    public static void main(String args[])  
    {  
        int b;  
        b = 'A';  
        System.out.write(b);  
        System.out.write('\n');  
    }  
}
```

- Вы не часто будете применять write () для выполнения консольного вывода (хотя это может быть полезно в некоторых ситуациях), потому что использовать print() println() намного легче.

# Класс PrintWriter

- Хотя использование объекта `System.out` для записи на консоль все еще допустимо в Java, его применение рекомендуется главным образом для отладочных целей или для демонстрационных программ.
- Для реальных Java-программ для записи на консоль рекомендуется работать с потоком типа `PrintWriter`.
- `PrintWriter` — это один из классов символьного ввода/вывода. Использование подобного класса для консольного вывода облегчает интернационализацию вашей программы.

## **`PrintWriter (OutputStream OutputStream, boolean flushOnNewline)`**

- Здесь **`OutputStream`** — объект типа `OutputStream`; **`flushOnNewline`** — **булевский** параметр, используемый как средство управления сбрасыванием выходного потока в буфер вывода (на диск) каждый раз, когда выводится символ `newline` (`\n`). Если `flushOnNewline` — `true`, поток сбрасывается автоматически, если — `false`, то не автоматически.
- `Printwriter` поддерживает методы `print ()` и `println()` для всех типов, включая `Object`.
- Чтобы записывать на консоль, используя класс `Printwriter`, создайте объект `System.out` для выходного потока, и сбрасывайте поток после каждого символа `newline`.
- Например, следующая строка кода создает объект типа `Printwriter`, который соединен с консольным выводом:

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

# Класс PrintWriter

```
// Демонстрирует Printwriter.  
import java.io.*;  
public class PrintWriterDemo  
{  
    public static void main(String args[])  
    {  
        Printwriter pw = new Printwriter(System.out, true);  
        pw.println("Это строка:");  
        int i = -7;  
        pw.println(i);  
        double d = 4.5e-7;  
        pw.println(d);  
    }  
}
```

- Вывод этой программы:

Это строка:

-7

4.5E-7

# Сериализация

- Сериализация – процесс преобразования объекта в последовательность байт. Сериализованный объект легко передать по сети, сохранить в файл.
- Что бы объект мог быть сериализован, он должен реализовывать интерфейс ***java.io.Serializable***, в котором нет ни одного метода.
- Классы ***ObjectInputStream*** и ***ObjectOutputStream*** производят сериализацию и десериализацию объектов.

# Пример

## Сериализация

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

## Десериализация

```
byte[] bArray =os.toByteArray();  
//получение содержимого массива  
  
ByteArrayInputStream is =new  
ByteArrayInputStream(bArray);  
  
ObjectInputStream ois =new  
ObjectInputStream(is);  
  
Object objRead =ois.readObject();
```

При этом объекты равны по значению, но не равны по ссылке!!!



# Восстановление объекта

- Сериализация объекта заключается в сохранении и восстановлении состояния объекта. Состояние описывается значением полей. Причем не только описанных в классе, но и унаследованных.
- При попытке самостоятельно написать механизм восстановления возникли бы следующие проблемы:
  1. Как установить значения полей, тип которых `private`
  2. Объект создается с помощью вызова конструктора. Как восстановить ссылку в этом случае
  3. Даже если существуют `set`-методы для полей, как выбрать значения и параметры.

# Способы сериализации

- Унаследовать класс от класса, который уже реализовал интерфейс ***Serializable***
- Если класс впервые реализовывает ***Serializable***, то у класса родителя должен быть конструктор без параметров, с помощью которого будут проинициализированы все унаследованные поля.

```
public class Parent{  
    public String firstname, lastname;  
    public parent(){ firstname='old_first'; lastname='old_last';}  
}
```

```
public class Child extends Parent implements Serializable{  
    private int age;  
    public Child (int age){ this.age=age;}  
}....
```

```
FileOutputStream fos=new FileOutputStream("output.bin");  
ObjectOutputStream oos=new ObjectOutputStream(fos);  
Child c= new Child(2);  
oos.writeObject(c); oos.close();
```

```
FileInputStream fis=new FileInputStream("output.bin");  
ObjectInputStream ois=new ObjectInputStream(fis);  
Ois.readObject(); ois.close();
```

# Результат

- При десериализации вызывается конструктор родительского класса, не реализующий `Serializable`. В итоге значение полей `firstname='old_first'`; `lastname='old_last'` осталось прежним (как в конструкторе `Parent`), а поле `age` – восстановилось и равно 2.
- Если не нужно сериализовать поле по какой-либо причине, то ставится модификатор `transient`

- Благодарю за внимание!
- Вопросы?