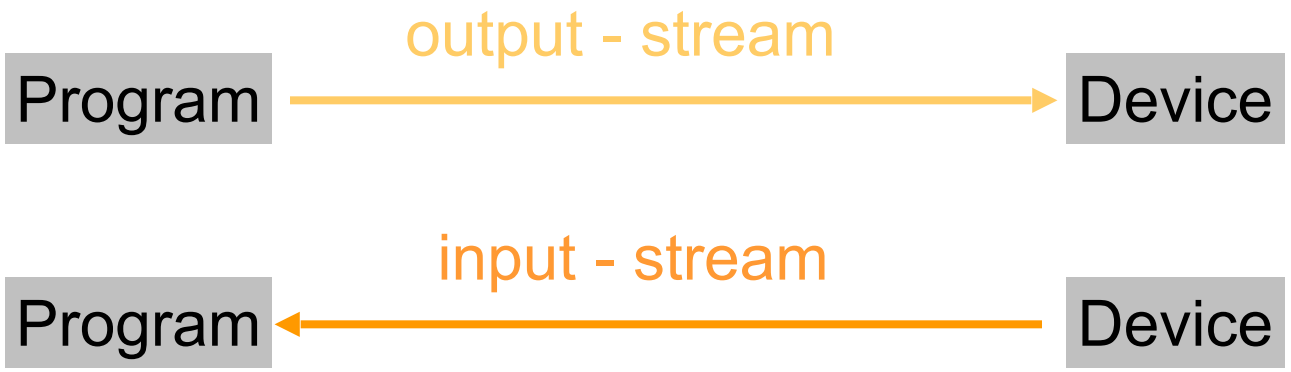
JAVA I/O:
Streams and Files

I/O

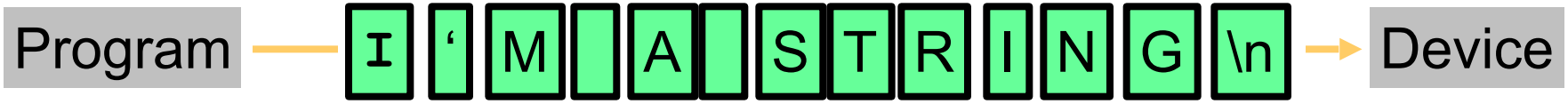
- Usual Purpose: storing data to ‘nonvolatile‘ devices, e.g. harddisk
- Classes provided by package java.io
- Data is transferred to devices by ‘streams‘



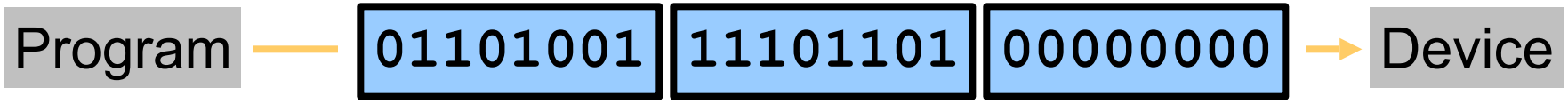
Streams

JAVA distinguishes between 2 types of streams:

- Text – streams, containing ‘characters‘



Binary Streams, containing 8 – bit information



Streams

Streams in JAVA are Objects, of course !

Having

- 2 types of streams (text / binary) and
- 2 directions (input / output)

results in 4 base-classes dealing with I/O:

1. Reader: text-input
2. Writer: text-output
3. InputStream: byte-input
4. OutputStream: byte-output

InputStream

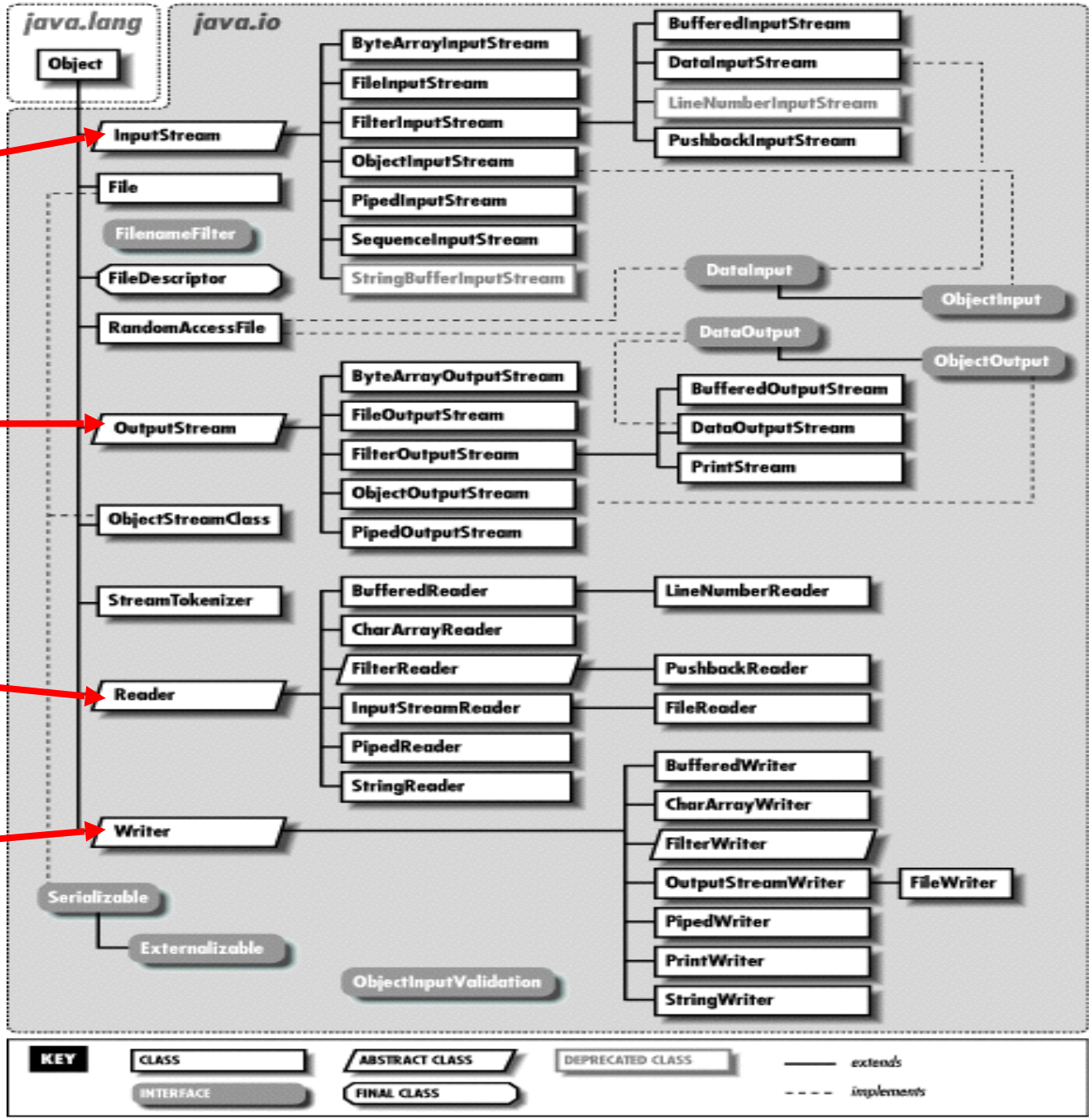
OutputStream

binary

Reader

Writer

text



Streams

- InputStream, OutputStream, Reader, Writer are abstract classes

- Subclasses can be classified by 2 different characteristics of sources / destinations:

  – For final device (data sink stream)
  purpose: serve as the source/destination of the stream
  (these streams ‘really’ write or read !)

  – for intermediate process (processing stream)
  Purpose: alters or manages information in the stream
  (these streams are ‘luxury’ additions, offering methods for convenient or more efficient stream-handling)

I/O: General Scheme

In General:

Reading (writing):

- open an input (output) stream
- while there is more information
  read(write) next data from the stream
- close the stream.

In JAVA:

- Create a stream object and associate it with a disk-file
  - Give the stream object the desired functionality
- while there is more information
  read(write) next data from(to) the stream
- close the stream.

Example 1

Writing a textfile:

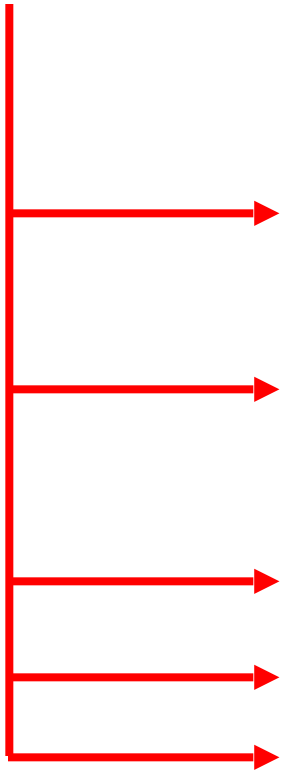
```
import java.io.*;

public class IOTest
{
    public static void main(String[] args)
    {
        try{

            FileWriter out = new FileWriter("test.txt");
            BufferedWriter b = new BufferedWriter(out);
            PrintWriter p = new PrintWriter(b);

            p.println("I'm a sentence in a text-file");

            p.close();
        }catch(Exception e){}
    }
}
```

- Create a stream object and associate it with a disk-file
- Give the stream object the desired functionality
- write data to the stream
- close the stream.

Writing Textfiles

Class: FileWriter

Frequently used methods:



Method Summary	
abstract void	**close**() Close the stream, flushing it first.
abstract void	**flush**() Flush the stream.
void	**write**(char[] cbuf) Write an array of characters.
abstract void	**write**(char[] cbuf, int off, int len) Write a portion of an array of characters.
void	**write**(int c) Write a single character.
void	**write**(String str) Write a string.
void	**write**(String str, int off, int len) Write a portion of a string.

Writing Textfiles

Using FileWriter

- is not very convenient (only String-output possible)
- Is not efficient (every character is written in a single step, invoking a huge overhead)

Better: wrap FileWriter with processing streams

- BufferedWriter
- PrintWriter

Wrapping Textfiles

BufferedWriter:

- Buffers output of FileWriter, i.e. multiple characters are processed together, enhancing efficiency

PrintWriter

- provides methods for convenient handling, e.g. println()

(remark: the System.out.println() – method is a method of the PrintWriter-instance System.out !)

Wrapping a Writer

A typical codesegment for opening a convenient, efficient textfile:

```
FileWriter out = new FileWriter("test.txt");
BufferedWriter b = new BufferedWriter(out);
PrintWriter p = new PrintWriter(b);
```

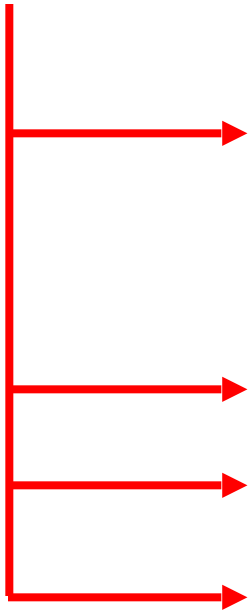
Or with anonymous (‘unnamed‘) objects:

```
PrintWriter p = new PrintWriter(
                    new BufferedWriter(
                      new FileWriter("test.txt")));
```

Reading Textfiles

Class: ReadText

Frequently used Methods:



Method Summary	
abstract void	**close**() Close the stream.
void	**mark**(int readAheadLimit) Mark the present position in the stream.
boolean	**markSupported**() Tell whether this stream supports the mark() operation.
int	**read**() Read a single character.
int	**read**(char[] cbuf) Read characters into an array.
abstract int	**read**(char[] cbuf, int off, int len) Read characters into a portion of an array.
boolean	**ready**() Tell whether this stream is ready to be read.
void	**reset**() Reset the stream.
long	**skip**(long n) Skip characters.

(The other methods are used for positioning, we don't cover that here)

Wrapping a Reader

Again:

Using FileReader is not very efficient. Better wrap it with BufferedReader:

```
BufferedReader br =
      new BufferedReader(
        new FileReader("name"));
```

Remark: BufferedReader contains the method readLine(), which is convenient for reading textfiles

EOF Detection

Detecting the end of a file (EOF):

- Usually amount of data to be read is not known
- Reading methods return ‘impossible‘ value if end of file is reached
- Example:
  - FileReader.read returns -1
  - BufferedReader.readLine() returns ‘null‘
- Typical code for EOF detection:

```
while ((c = myReader.read() != -1){  // read and check c
    ...do something with c
}
```

Example 2: Copying a Textfile

```
import java.io.*;
public class IOTest
{
        public static void main(String[] args)
        {
            try{
                        BufferedReader myInput = new BufferedReader(new
                                FileReader("IOTest.java"));
                        BufferedWriter myOutput = new BufferedWriter(new
                                FileWriter("Test.txt"));

                        int c;
                        while ((c=myInput.read()) != -1)
                                myOutput.write(c);

                        myInput.close();
                        myOutput.close();
            }catch(IOException e){}
        }
}
```

Binary Files

- Stores binary images of information identical to the binary images stored in main memory
- Binary files are more efficient in terms of processing time and space utilization
- drawback: not ‘human readable‘, i.e. you can‘t use a texteditor (or any standard-tool) to read and understand binary files

Binary Files

Example: writing of the integer ’42‘

- TextFile: ‘4‘ ‘2‘ (internally translated to 2 16-bit representations of the characters ‘4‘ and ‘2‘)
- Binary-File: 00101010, one byte

  (= 42 decimal)

Writing Binary Files

Class: FileOutputStream

... see FileWriter

The difference:

No difference in usage, only in output format

Reading Binary Files

Class: FileInputStream

... see FileReader

The difference:

No difference in usage, only in output format

Binary vs. TextFiles

	pro	con
Binary	Efficient in terms of time and space	Preinformation about data needed to understand content
Text	Human readable, contains redundant information	Not efficient