# File Class

- File Class is used to
  - Examine files and directories
  - Examine the list of files or directories on the hard drive
  - Create new files and directories on the hard drive
  - Create file stream objects
- There are four constructors:

```
File myDir = new File(“C:\Temp”);

File myFile = new File(“C:\Temp\file.txt”);

File myFile = new File (myDir, “file.txt”);

File myFile = new File(“C:\Temp”, “file.txt”);
```

## Testing and Checking File Objects

- There are more than 30 methods that you can apply to File objects. These are a few:
  - `getName()`
  - `getPath()`
  - `getParent()`
- Query Files and Directory
  - `exists()`
  - `isDirectory()`
  - `isFile()`
- Creating and Modifying Files and Directories
  - `renameTo(File path)`
  - `mkdir()`
  - `createNewFile()`

# File Class Example

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

       File myDir = new File("C:/JavaTemp");
       System.out.println(myDir + (myDir.isDirectory() ? " is " : " is not ")
       + "a directory");

       File newDir = new File(myDir, "newDir");
       newDir.mkdir();


       String[] fileList = myDir.list();
       for (String f : fileList) {
           System.out.println(f);
       }
   }
```

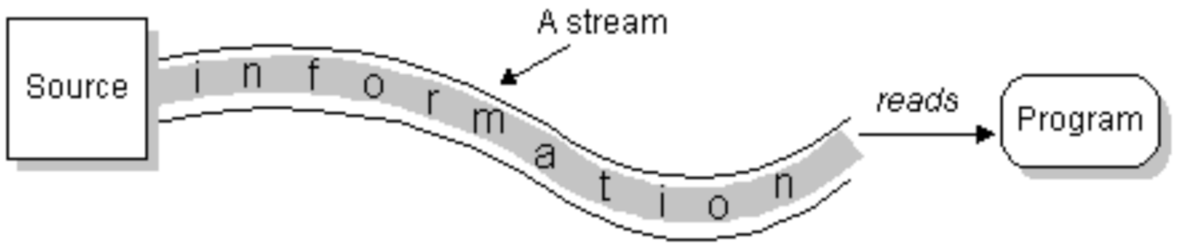
Is JavaTemp a directory in C:\ drive?

Create a directory called “newDir” in C:\JavaTemp folder

list of files/directories in C:\JavaTemp

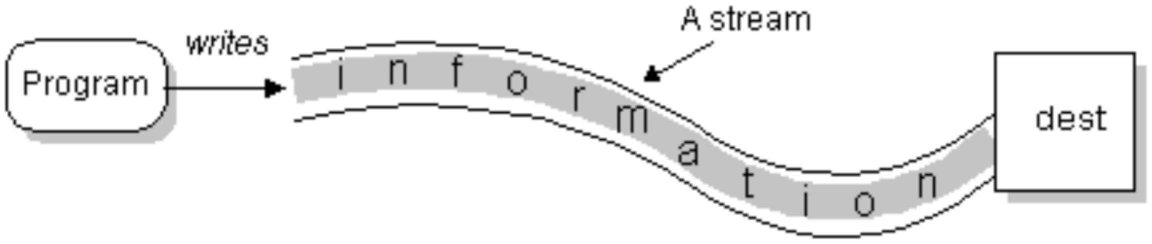
## Input Stream

- To bring in information, a program opens a **stream** on the source (a file, memory, a socket) and reads the information sequentially:



# Output Stream

- To output information, a program opens a **stream** to the destination (a file, memory, a socket) and writes the information sequentially:



## IO Stream

- No matter where the data is coming from or going to and no matter what its type, the algorithms for sequentially reading and writing data are :

**Reading**

```
open a stream
while more information{
   read information
}
close the stream
```

**Writing**

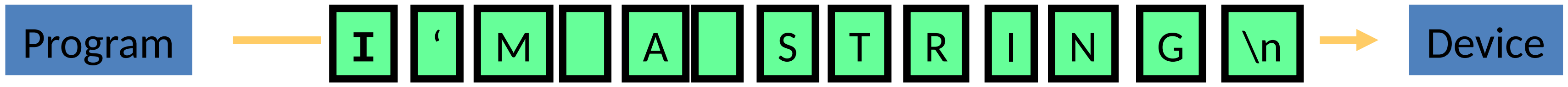
```
open a stream
while more information{
   write information
}
close the stream
```

## Character and Binary Streams

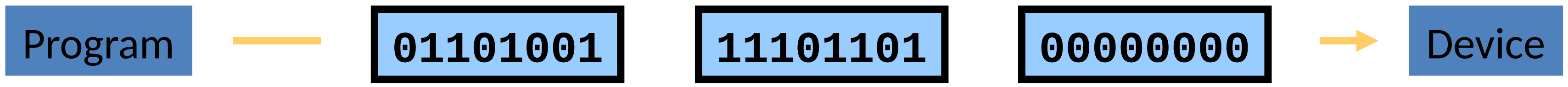
- The stream classes are divided into two class hierarchies, based on the data type:
  - Character Stream
    - Java stores its characters internally as 16-bit Unicode characters (2 bytes)
    - Characters streams are used for storing and retrieving text
  - Binary Stream
    - A series of bytes exactly as it appears in memory
    - No transformation of data takes place

# Character and Binary Streams

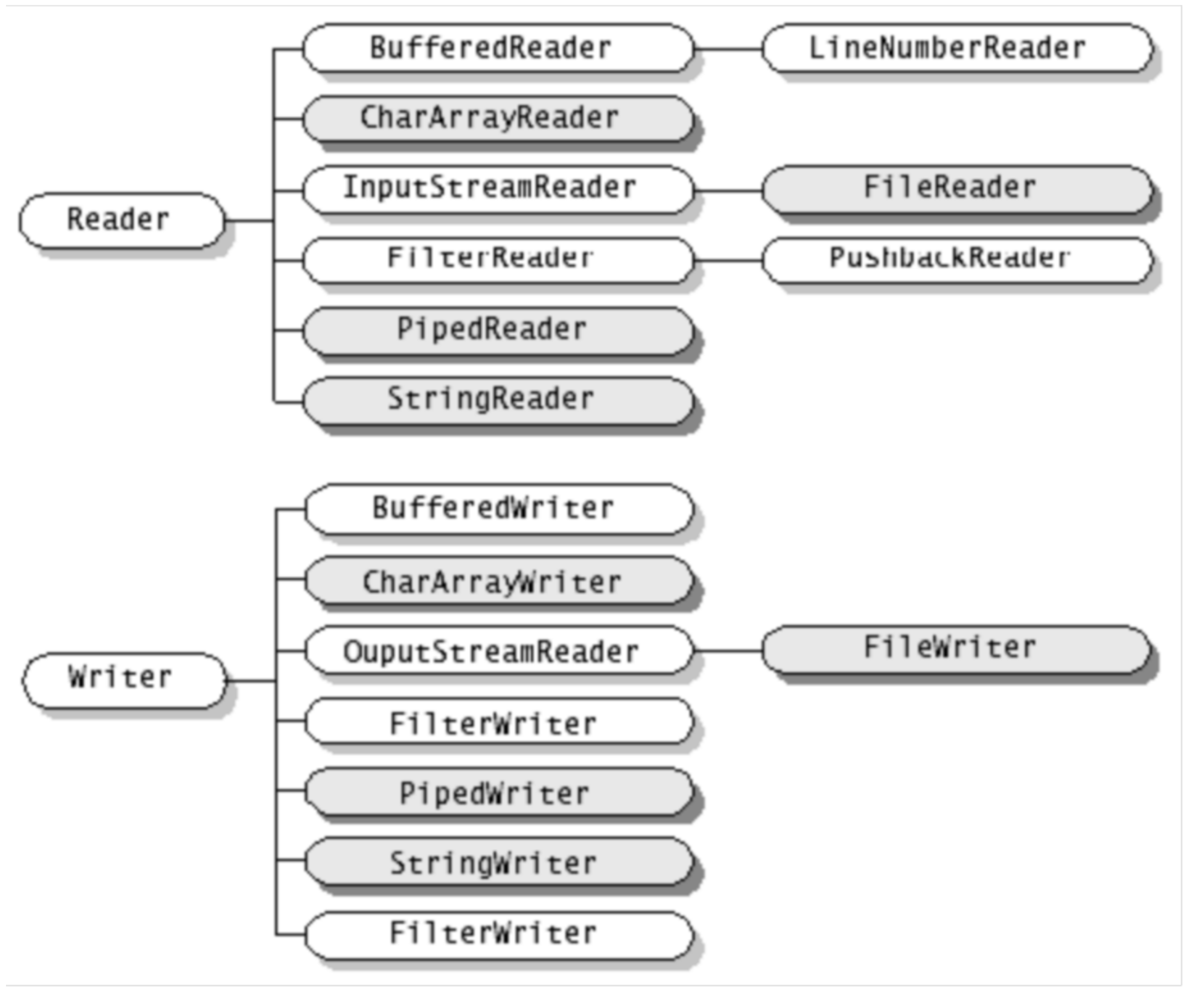
Character Stream – streams, containing ‘text‘



Binary Streams, containing 8 – bit information



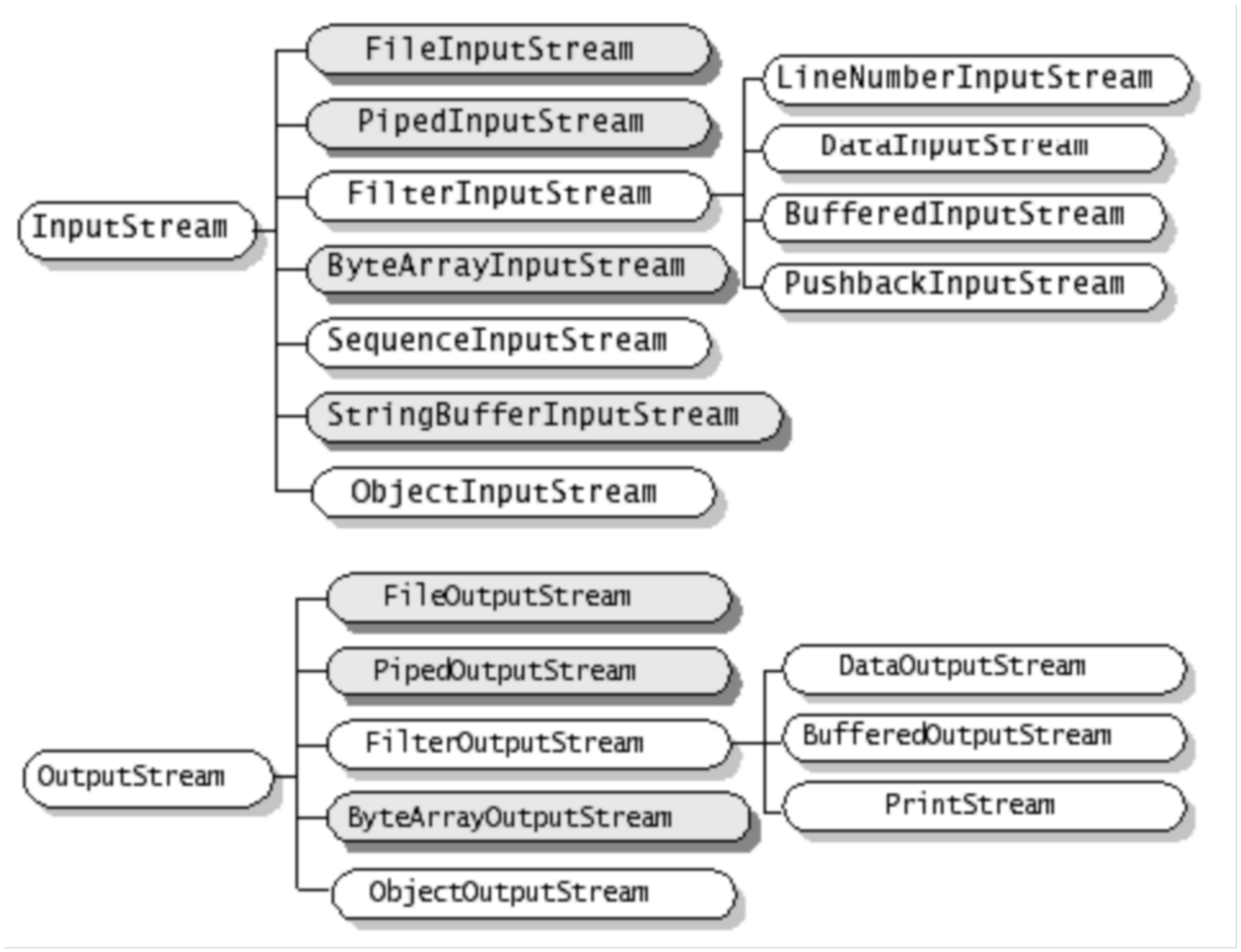
# Character Stream Class Hierarchy



## Stream Reader and Writer

- `Reader` and `Writer` are the abstract superclasses in `java.io` for 16 bit character streams
  - `Reader` provides the methods and partial implementation for *readers*
  - `Writer` provides the methods and partial implementation for *writers*
- Subclasses of `Reader` and `Writer` implement specialized streams and are divided into two categories
- Most programs should use *readers* and *writers* to read and write textual information

## Binary Stream Class Hierarchy



## Input/Output Binary Stream

- `InputStream` and `OutputStream` are the abstract superclasses in `java.io` for 8 bit byte streams
  - `InputStream` provides the methods and partial implementation for *input streams*
  - `OutputStream` provides the methods and partial implementation for *output streams*
- Subclasses of `InputStream` and `OutputStreams` implement specialized streams and are divided into two categories

- Most programs should use *input streams* and *output streams* to read and write byte information such as images, sounds, etc

## Character and Binary Stream Methods

- **Reader** and **InputStream** define similar abstract methods but for different data types
  - **Reader** contains these methods for reading characters and arrays of characters:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

  - **InputStream** defines the same methods but for reading bytes and arrays of bytes

```
int read()
int read(byte cbuf[])
int read(byte cbuf[], int offset, int length)
```

## Character and Binary Stream Methods

- **Writer** and **OutputStream** define similar abstract method but for different data types
  - **Writer** contains these methods for writing characters, arrays of characters, and strings:

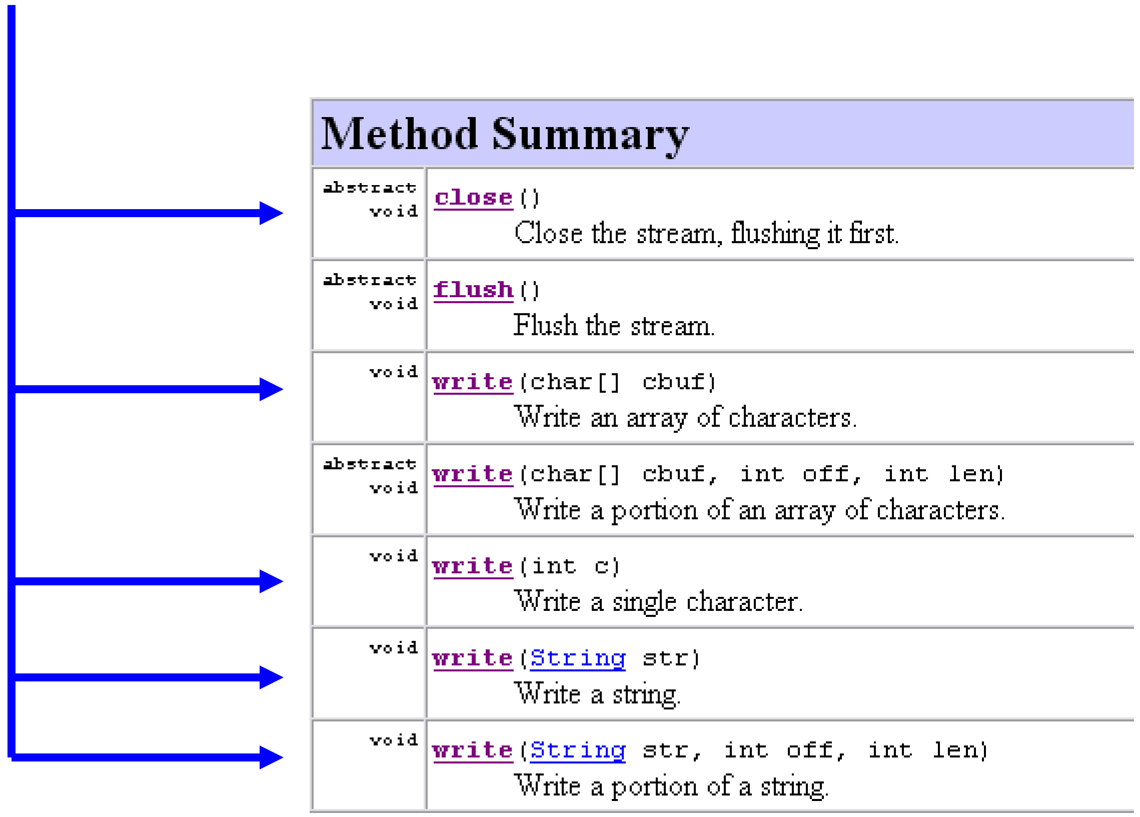
    ```
        void write(int c)
    void write(char cbuf[])
    void write(char cbuf[], int offset, int length)
    ```

  - **OutputStream** defines the same methods but for writing bytes and arrays of bytes

    ```
        void write(int c)
    void write(byte cbuf[])
    void write(byte cbuf[], int offset, int length)
    ```

# 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:

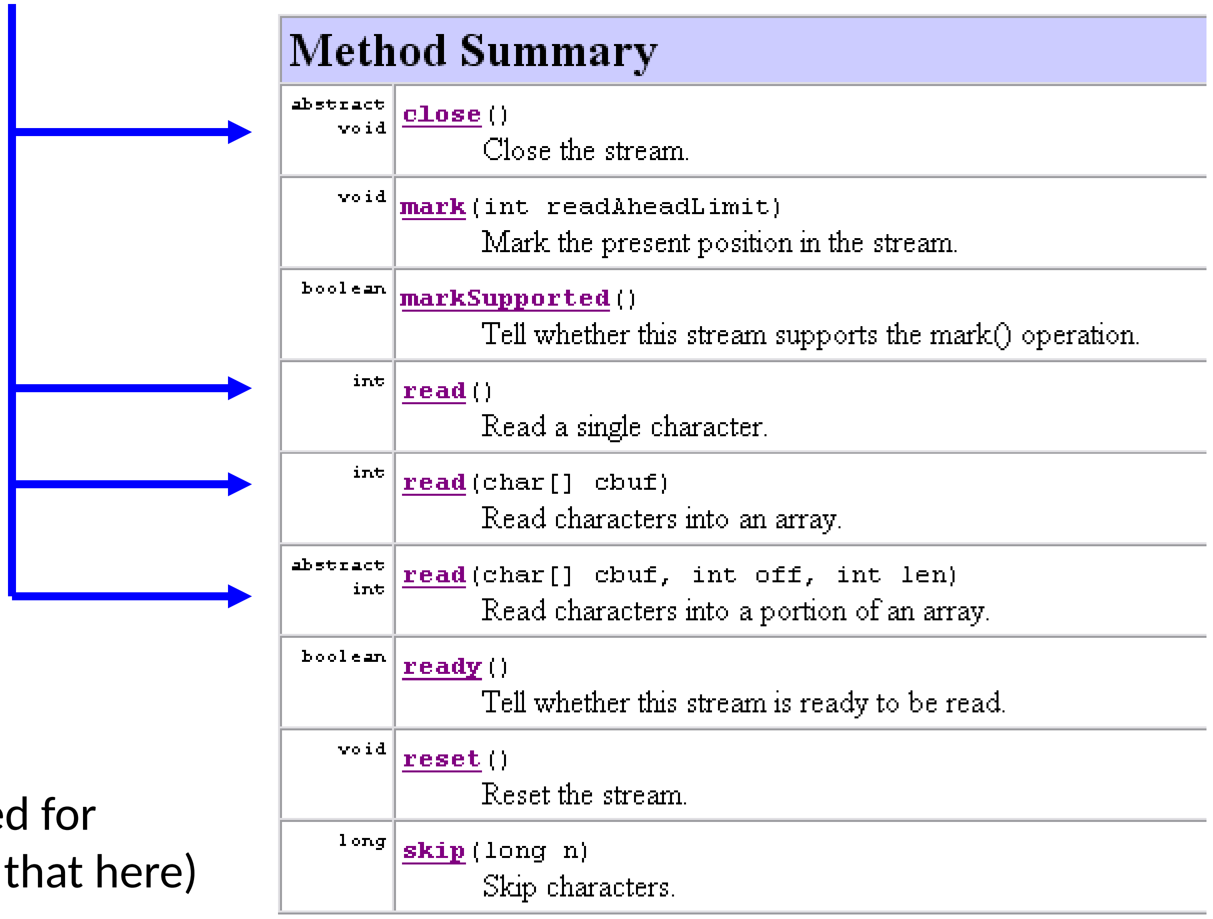
```
FileWriter out = new FileWriter("test.txt");
BufferedWriter b = new BufferedWriter(out);
PrintWriter p = new PrintWriter(b);
```

**OR**

```
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“));
```

# 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: 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 |