Java I/O

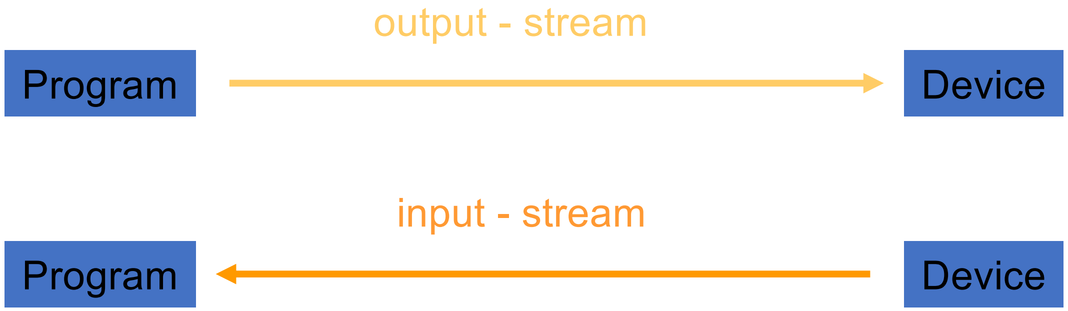
Streams

I/O in Java

- Usual purpose:
  - Storing data to ‘nonvolatile‘ devices, e.g. harddisk
  - Reading data from ‘nonvolatile‘ devices

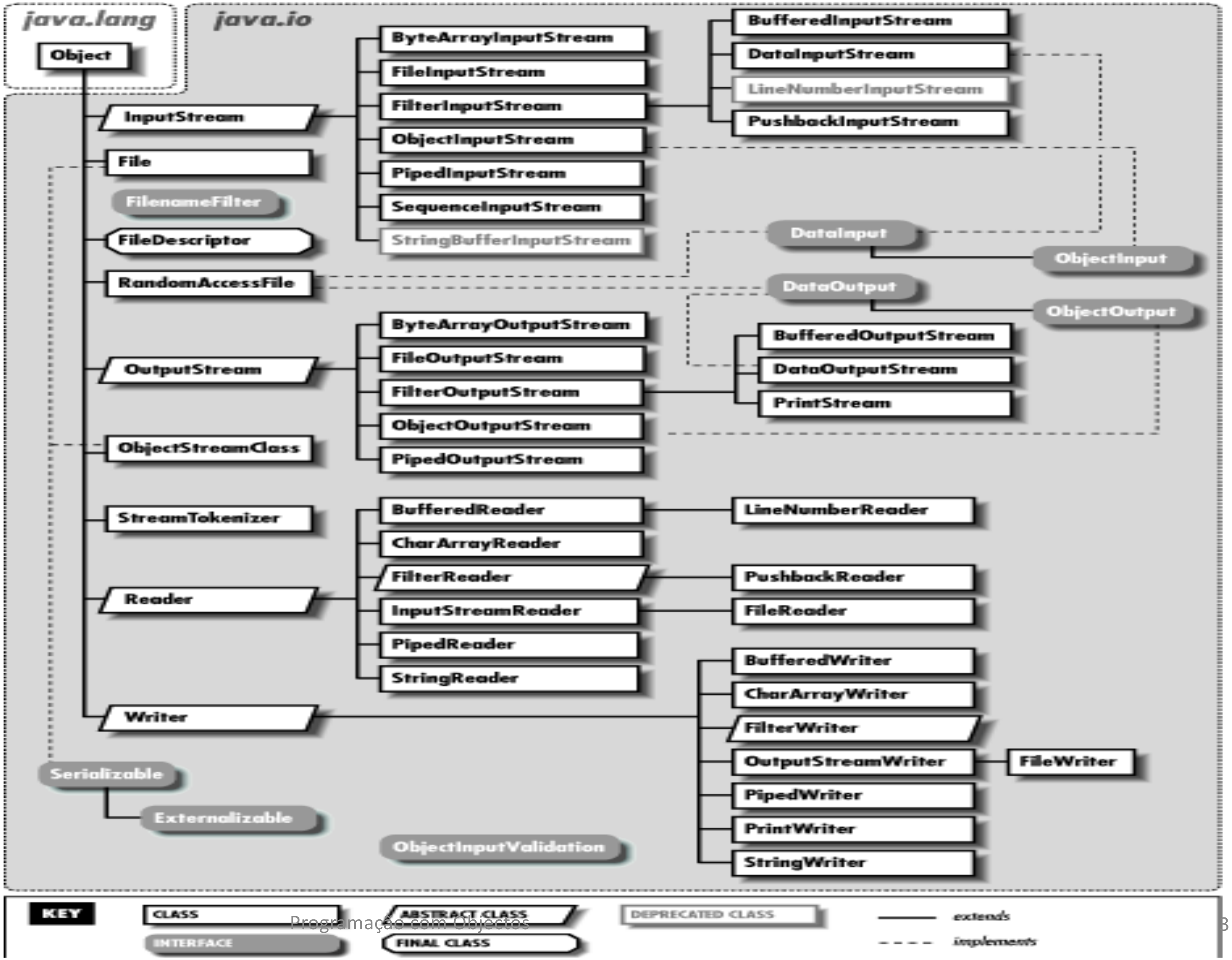
- Classes provided by package java.io

- Data is transferred to/from devices by ‘streams‘



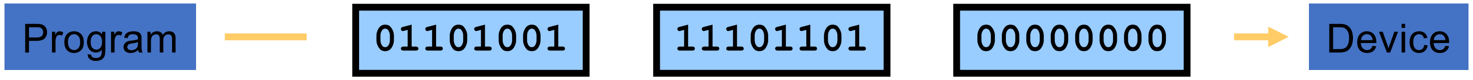
I/O in Java

- Implemented in java.io
- Based on the concept of Stream

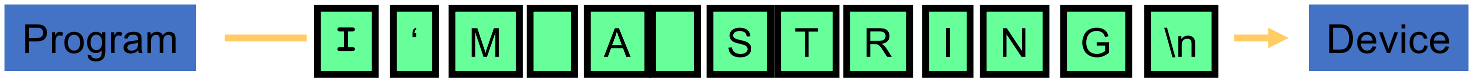


Those Scary Stream Classes

- Most programmers are taken aback by the complexity of the stream classes
  - There are many classes in the java.io package
  - The applicability of each class is not always obvious
- To deal with the complexity of the java.io, Java considers 3 dimensions:
  - Input and Output oriented streams
  - Type of device
    - File, socket, …
  - Content of the stream
    - Byte-oriented (binary)



    - versus Character-oriented (text)



IO in Java: Streams

- Streams in Java are Objects, of course!

- Have a problem with four possible combinations:
  - 2 types of streams (text / binary) and
    - text = character
  - 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
  - All abstract

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 !)

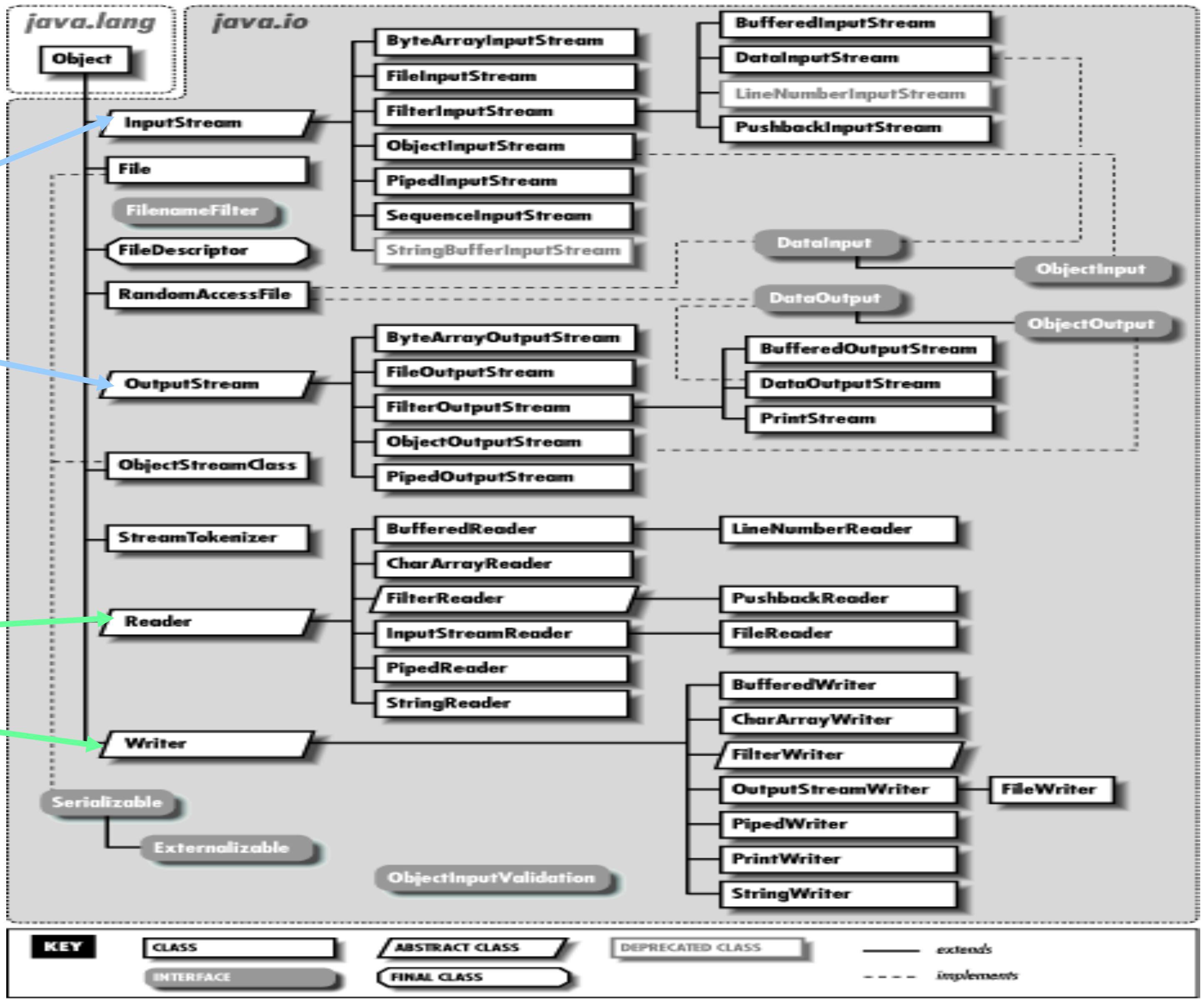
  - For intermediate process (processing stream)

    Purpose: alters or manages information in the stream

    (these streams are 'luxury' additions, offering methods for convenient or more efficient stream-handling)

I/O in Java

- Now is easy!



I/O: General Scheme

- General I/O processing:

  Reading (writing):

  1. open an input (output) stream
  2. while there is more information

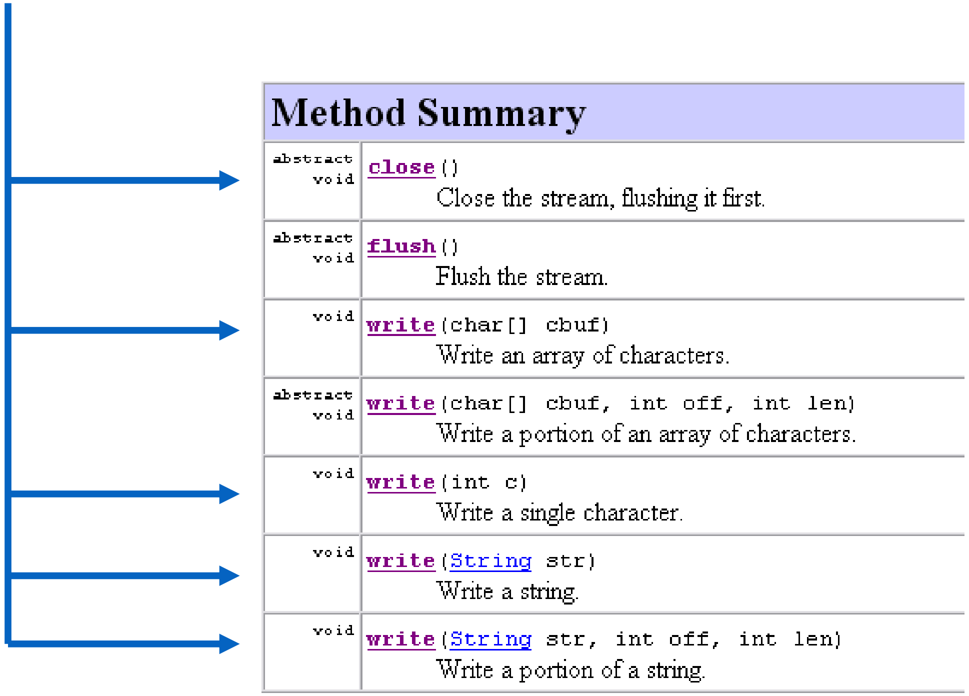
     read(write) next data from the stream
  3. close the stream

- In Java:

  1. create a stream object and associate it with a sink/source
  2. give the stream object the desired functionality
  3. while there is more information
     1. read(write) next data from(to) the stream
  4. close the stream

Writing Text Files

- 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 Text Files

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

- Better Solution: 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()
  - Both print() and println() are overloaded to take a variety of types
  - System.out and System.err are PrintWriters

Wrapping a Writer

- A typical code segment for opening a convenient, efficient text file:

```
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")));
```

Example 1 – Writing to a file

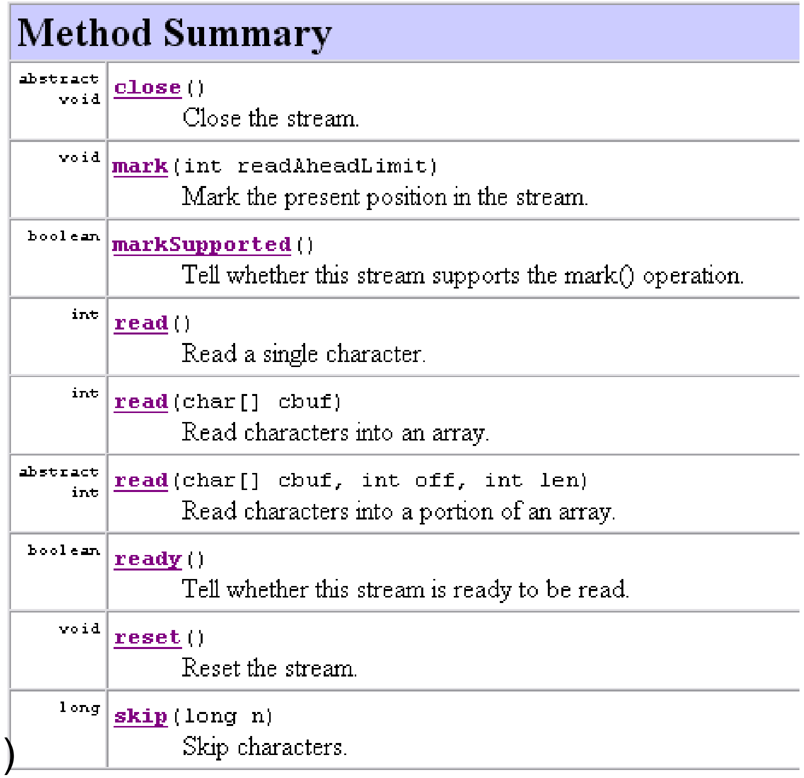
- Writing a text file:

```
public class WriteCharacters {
  public static void main(String[] args) throws IOException {
    FileWriter out = null;
    BufferedWriter bout = null;
    try {
      out = new FileWriter("characteroutput.txt");
      bout = new BufferedWriter(out);
      bout.write(“Writing something”);
    } finally {
      if (bout != null)
        bout.close();
    }
  }
}
```

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

Reading Textfiles

- Class: FileReader
- 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.

- Anything strange?
  - **int** read()

(The other methods are used for

positioning, we don't cover that here)

Wrapping a Reader

- Again: Using FileReader is not very efficient.
- Better Solution:
  - Wrap it with BufferedReader:
- Example

```
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(args[0]));
          BufferedWriter myOutput = new BufferedWriter(new FileWriter(args[1])); )  {
      int c;
      while ( (c = myInput.read())  !=  -1) {
        myOutput.write(c);
      }
    } catch (IOException e) {
       System.out.println(„Error while copying “ + e.getMessage());
       e.printStackTrace();
    }
  }
}
```

- No need to close streams. Why?

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 text editor (or any standard-tool) to read and understand binary files
- Because they are byte-oriented, they are inflexible when dealing with multi-byte characters
    - Byte oriented streams only directly support ASCII
    - International fonts would require extra work for the programmer

Binary Files

Example: writing of the integer ’42‘

- Text File: ‘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

- Implemented by FileOutputStream
- Main methods
  - close()
  - write(byte[] b)
  - write(byte[] b, int off, int len)
  - write(int b)

- Similar to FileWriter
  - No difference in usage, only in input format

Reading Binary Files

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

Binary vs. Text Files

- When to use Text / Binary Files ?
- **ALWAYS** use Text Files for final results
    - Unless there is an imperative reason to favor efficiency against readability.
- Binary Files might be used for non-final interchange between programs
- Binary Files are always used for large amount of data (images, videos etc.),
    - but there's always an *exact* definition of the meaning of the byte stream
        - Example: JPG, MP3, BMP

Conversion

- Character oriented streams can be used in conjunction with byte-oriented streams:
- Use InputStreamReader to "convert" an InputStream to a Reader
- Use OutputStreamWriter to "convert" Writer into an OutputStream
- It is possible to specify the character encoding to apply.

Object Serialization

- When an object is instantiated, the system reserves enough memory to hold all of the object's instance variables
  - The space includes inherited instance variables
- The object exists in memory
  - Instance methods read and update the memory for a given object.
- The memory which represents an object can be written to an **ObjectOutputStream**
  - Responsible for converting an object into byte[]
  - And then writing the array into an OutputStream

Object Serialization - 2

- What about the objects referenced by a serializable object?
    - All **non-static** fields of an object are serializable
    - Any other objects referred to by the serialized object are also serialized to the stream
        - Unless they are marked as *transient*
    - Objects are not **duplicated** when serialized
- Serializable classes must implement the **java.io.Serializable** interface
    - When an object is serialized, the stream checks this
        - If not, the Stream throws a NotSerializableException
    - The Serializable interface does not define any methods
    - Define field in serializable class **static final long serialVersionUID = someValue**;
        - Used to know if version of class is compatible with serialized object

Object Serialization - 3

- Serialize an object: Use a ObjectOutputStream and an OutputStream
- Main methods:
  - **writeInt**(int)
  - **writeFloat**(float)
  - **writeObject**(Object)
  - …
- How to do the inverse operation (Convert bytes into an object)?
  1. Use ObjectInputStream and an InputStream
  2. Similar available methods for reading

Example - Serialize an Object

```
import java.io.*;
public class Test {
   public void saveObject(String file, Object obj)  throws IOException {
      ObjectOutputStream obOut = null;
      try {
        obOut = new ObjectOutputStream(new FileOutputStream(file));
        obOut.writeObject(obj);
      } finally {
        if (obOut != null)
          obOut.close();
      }
   }
}
```

With try-with-resources version

```
import java.io.*;
public class Test {
   public void saveObject(String file, Object obj) throws IOException {
     try (ObjectOutputStream obOut =
           new ObjectOutputStream(new FileOutputStream(file))) {
           obOut.writeObject(obj);
     }
   }
}
```

Example - Read in a Serialized Object

```
import java.io.*;

public class Test {
  public Object readObject(String inputFilename) throws IOException  {
    ObjectInputStream objIn = null;
    try {
      objIn = new ObjectInputStream(new FileInputStream(inputFilename));
      Object anObject = obIn.readObject();
      return anObject;
    } finally {
      if (objIn != null)
        objIn.close();
    }
  }
}
```

- Can simplify with try-with-resources

Example - Serialize an Object and Compress

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

public class Test {
 public void saveObject(String filename, Object obj) throws IOException {
  ObjectOutputStream obOut = null;
  try {
   FileOutputStream fpout = new FileOutputStream(filename);
   DeflaterOutputStream dOut = new DeflaterOutputStream(fpout);
   obOut = new ObjectOutputStream(dOut);
   obOut.writeObject(obj);
  } finally {
   if (obOut != null)
    obOut.close();
  }
 }
}
```

Example - Read in a Compressed Serialized Object

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

public class Test {
  public Object readObject(String inputFilename) throws IOException {
    ObjectInputStream obIn = null;
    try {
      FileInputStream fpin = new FileInputStream(inputFilename);
      InflaterInputStream inflateIn = new InflaterInputStream(fpin);
      obIn = new ObjectInputStream(inflateIn);
      Object anObject = obIn.readObject();
      return anObject;
    } finally {
      if (obIn != null)
        obIn.close();
    }
  }
}
```