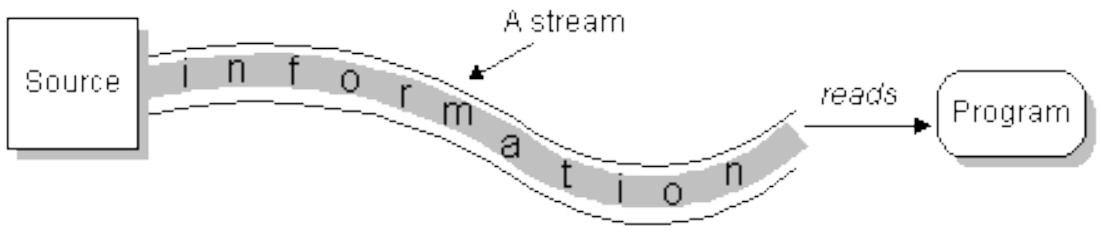
Java Input/Output Streams

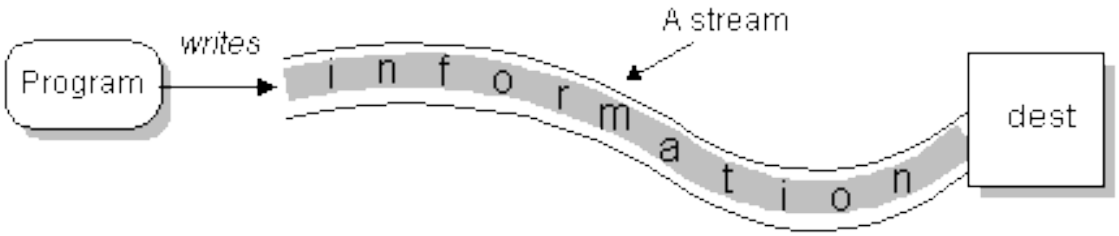
Rui Moreira

Some useful links:
http://java.sun.com/docs/books/tutorial/essential/TOC.html#io

Input Stream



Output Stream



JVM creates the streams

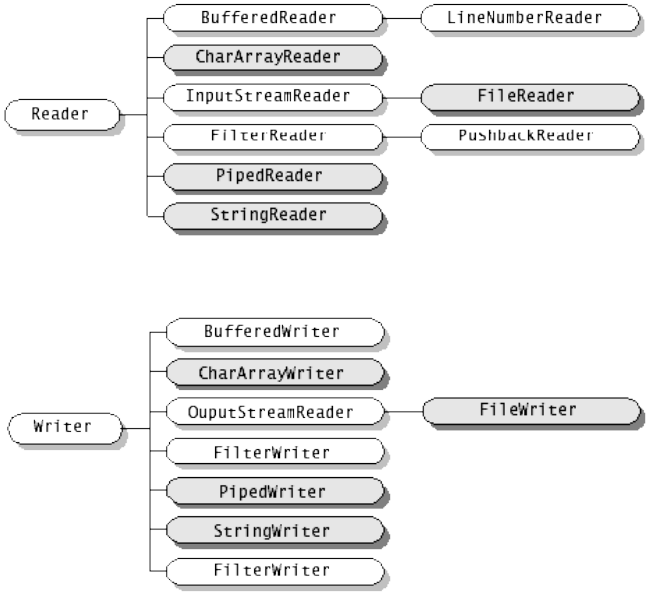
- **System.in** (type InputStream): **stdin**
  (keyboard input)
- **System.out** (type PrintStream): **stdout**
  (keyboard output)
- **System.err** (type PrintStream): **stderr**
  (keyboard output for error messages)

Input / Output

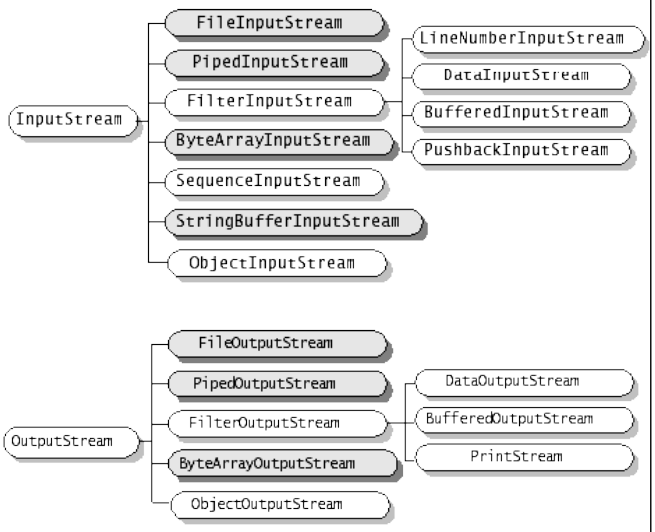
- Internally Java uses Unicode for encoding characters, i.e., 2 bytes for each character which permits a larger set than ASCII;
- Java 1.1 uses 16-bit characters but to remain compatible with previous (Java 1.0) code it accepts 8-bit streams for keyboard I/O;
- There are several adapter classes (e.g. `InputStreamReader`, `OutputStreamWriter`) to convert 8-bit streams to 16-bit;
- There are several wrappers for binary input which read bytes and then transform/cast values according to the data types involved.

Classes from **java.io** package

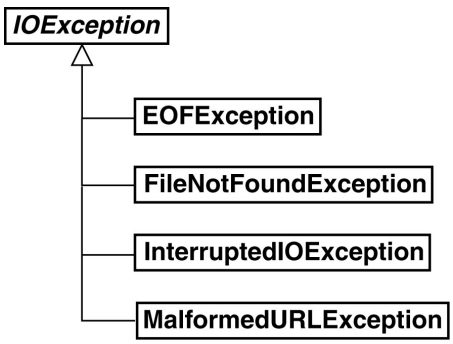
Char streams - can handle **text** files - any char in Unicode set (16-bit)



Byte streams - handle **binary** and **text** files - but limited to ISO Latin 1 (8-bit)



Exception Classes



File class (getting **file** info)

```
import java.io.*;

public class FileApp {
  public static void main(String args[]){
    String filename = (args.length==1?args[0]:"Teste.txt");
    try {
      // Open file inside "classes" folder
      File file = new File("classes", filename);
      // Print some file information
      System.out.println("File name - " + file.getName());
      System.out.println("File path - " + file.getPath())
      System.out.println("File Size - " + file.length());
      System.out.println("File Absolut Path - " + file.getAbsolutePath());
      Date d = new Date(file.lastModified());
      System.out.println("File Last Modified - " + d.toString());
      System.out.println("File is dir? - " + file.isDirectory());
      System.out.println("File is file? - " + file.isFile());
      System.out.println("File is hidden? - " + file.isHidden());
    } catch (Exception e) {
        System.err.println("FileApp - main(): "+e.toString());
    }
}
```


File class (getting **folder** info)

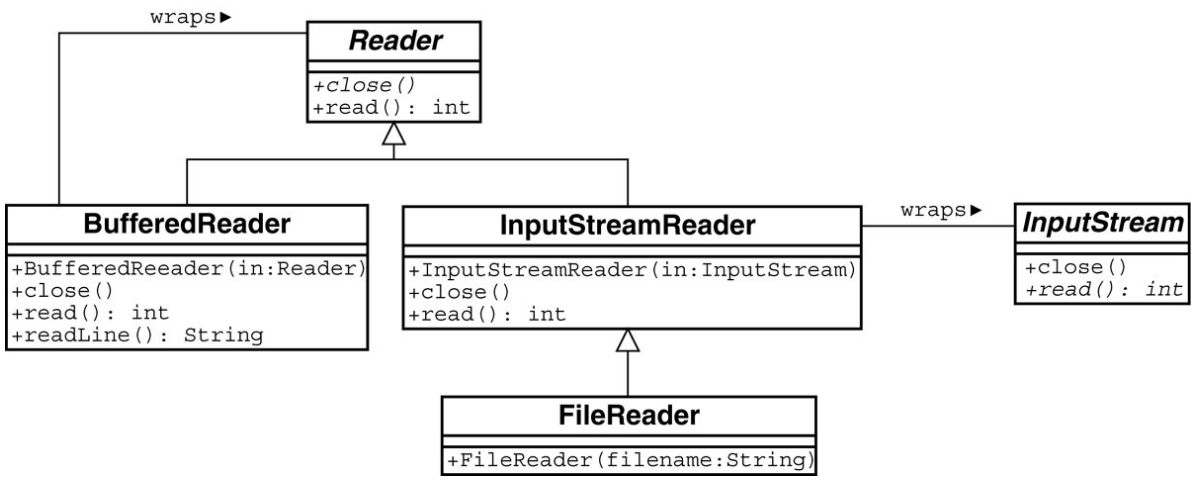
```
import java.io.*;

public class FileApp {
  public static void main(String args[]){
    String filename = (args.length==1?args[0]:"Teste.txt");
    try {
      // Open "classes" folder
      File file = new File("classes");
      // Print folder information
      System.out.println("\nFileApp - main(): folder info:");
      System.out.println("Folder name - " + folder.getName());
      System.out.println("Folder is directory - " + folder.isDirectory());
      System.out.println("Folder path - " + folder.getPath());
      System.out.println("Folder can read - " + folder.canRead());
      System.out.println("Folder can write - " + folder.canWrite());
      System.out.println("Folder is hidden - " + folder.isHidden());
      System.out.println("Folder content - ");
      // Print folder content
      String[] content = folder.list();
      for(int f=0; f<content.length; f++) System.out.println(" "+content[f]);
    } catch (Exception e) {
        System.err.println("FileApp - main(): "+e.toString());
    }
}
```

Input from File (FileInputStream & DataInputStream)

```
import java.io.*;

public class DataInputStreamApp {
  public static void main(String args[]){
    String filename = (args.length==1?args[0]:"Test.txt");
    String line="";
    int line_count = 0;
    try {
      // Open byte stream
      FileInputStream fis = new FileInputStream(filename);
      DataInputStream dis = new DataInputStream(fis);
      System.out.println("DataInputStreamApp - main(): file lines are...");
      // THE readLine() METHOD IS DEPRECATED, THEREFORE,
      // WITH TEXT FILES USE Reader/Writer Classes INSTEAD (see next slides)
      while ((line = dis.readLine()) != null) {
        System.out.println(">"+line_count+" - "+line);
        line_count++;
      }
      dis.close();
      } catch (Exception e) {
         System.err.println("DataInputStreamApp - main(): file input error!");
      }
}
```


Input Classes: **java.io** package

Read from stream...



Text Input Classes

- `FileReader fr = FileReader(filename);`

  open a file for reading one **char** at a time

- `BufferedReader br = new BufferedReader(new FileReader(filename));`

  read a **block of characters** at a time (buffering); treat files as streams of characters (increases reading efficiency and allows line-based reading).

Text Input - FileReader class

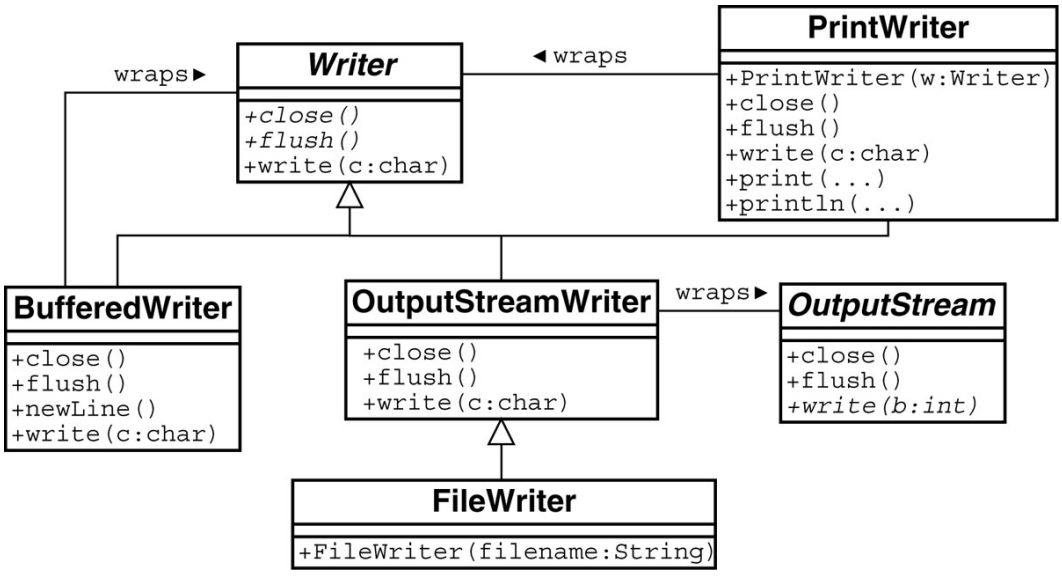
```
import java.io.*;

public final class FileReaderApp {

    public static void main (String[] args) {
       FileReader fr = null;
       try {
            String filename = (args.length==1?args[0]:"Test.txt");
            // The Output folder is the Working Directory defined
            // in the JBuilder Project Properties
            fr = new FileReader(filename);
            int i=0;
            while ((i=fr.read())!=-1) System.out.print((char)i);
       } catch (IOException ioe) {
            ioe.printStackTrace();
       } finally {
            try {
                 if (fr!=null) fr.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
       }
   }
}
```

Text Input - BufferedReader class

```
import java.io.*;

public final class BufferedInputApp {

  // Does not handle IOException - throws it
  public static void main (String[] args) throws IOException {

    String filename = (args.length == 1 ? args[0] : "Test.txt");
    String line = "";
    File file = new File(filename);
    FileReader fr = new FileReader(file);
    BufferedReader br = new BufferedReader(fr);

    // Read lines from file
    int line_count = 1;
    while ((line = br.readLine()) != null) {
      System.out.println("Linha " + (line_count++) + " = " + line);
    }
  }
}
```


Output Classes: **java.io** package

Write to stream...



Text Output Classes

- `FileWriter fw = FileWriter(filename);`

  open a file for writing one **char** at a time

- `PrintWriter pr = new PrintWriter(new FileWriter(filename));`

  open a file for printing **chars** or **lines**

- `BufferedWriter bw =`
  `new BufferedWriter(new FileWriter(filename));`

  write a **block of characters** at a time (buffering);

  treat files as streams of characters (increases writing efficiency and allows line-based writing

Text Output - PrintWriter class

```
import java.io.*;

public class PrintWriterApp {

  public static void main(String args[]) {
    try {
      // We can use either FileOutputStream or FileWriter
      //FileOutputStream fos = new FileOutputStream("classes\\Test.txt");
      //PrintWriter pw = new PrintWriter(fos);
      FileWriter fw = new FileWriter("classes\\Test.txt");
      PrintWriter pw = new PrintWriter(fw);

      // Print into to the file
      pw.println("Hello world!");
      // Flush and close
      pw.flush();
      pw.close();
    } catch (Exception e) {
      System.err.println("PrintWriterApp - main(): "+e.toString());
    }

  }
}
```

Sequential Binary Files

- Open a file for sequential writing, i.e., reading or writing the file from beginning to end as a sequential stream of chars/bytes

- We can store (read/write) both primitive values and objects:

  - **Primitive** types (e.g., byte, short, int, long, float, double, etc.)

  - **Object** types (e.g., String, Date, Client, Account, etc.)

Primitive Binary Output (DataOutputStream)

```
import java.io.*;

public class BinOutputFileApp {
  public static void main(String args[]){
    try {
         String filename = (args.length == 1?args[0]:"Test.bin");
         double data[] = {1.3, 1.6, 2.1, 3.3, 4.8, 5.6, 6.1, 7.9, 8.2, 9.9};
         // Open file and wrappers
         File file = new File(filename);
         FileOutputStream fos = new FileOutputStream(file);
         DataOutputStream dos = new DataOutputStream(fos);

         // Write doubles into the file
         for (int i = 0; i < data.length; i++) {
                 dos.writeDouble(data[i]);
         }
         dis.close();

    } catch (Exception e) {
         e.printStackTrace();
    }
  }
}
```

Primitive Binary Input (DataInputStream)

```
import java.io.*;

public class BinInputFileApp {

  public static void main(String arg[]){
    try {
         String filename = (args.length == 1 ? args[0] : "Test.bin");
         FileInputStream fis = new FileInputStream(filename);
         DataInputStream dis = new DataInputStream(fis);

         // Read doubles (binary data) from file; There are other methods:
         // readByte(), readInt(), readLong(), readShort(), readFloat(), etc.
         while (dis.available()!=0) {
           double d = dis.readDouble();
           System.out.println("FileInputApp - main(): " + d);
         }
         dis.close();

    } catch (Exception e) {
         e.printStackTrace();
    }
  }
}
```


Interface java.io.Serializable

- **Interface** with no methods - is a "**marker/tag**" indicating to the JVM that a given class can be serialized, i.e., become persistent
- Persistency - save object to **stream of bytes**, i.e., permanent storage, net transmition,etc
- Classes that do not implement Serializable cannot save/restore their state (member variables only)
- **Only** the **data of objects** is preserved (**made peristency**) – class **methods** and **constructors** are **not** part of the serialized stream

Interface java.io.Serializable

- The JVM serializes the **entire object graph**, i.e., when serializing na object all the inner-objects are serialized too
- An object that contains **non-serializable inner-objects** (member variables of non-serializable class types) cannot be serialized – launches **exception** - **`NotSerializableException`**
- It is possible to **mark/declare** a member variable not to be serialized (`private` **`transient`** `Thread t;`) and avoid exceptions (thus allowing serialization of outer-object)

Object Binary Output (ObjectOutputStream)

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

public class ObjectOutputApp {
  public static void main(String arg[]){
    try {
         String filename = (args.length==1?args[0]:"classes\\Date.bin");
         File f = new File(filename);
         FileOutputStream fos = new FileOutputStream(f);
         ObjectOutputStream oos = new ObjectOutputStream(fos);
         // Object Date - today's date
         Date today = new Date(System.currentTimeMillis());
         // Write String object followed by Date object
         oos.writeObject("Today's Date:");
         oos.writeObject(today);
         oos.flush();
         oos.close();
    } catch (Exception e) {
         e.printStackTrace();
    }
  }
}
```

Object Binary Input (ObjectInputStream)

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

public class ObjectInputApp {
  public static void main(String arg[]){
    try {
         String filename = (args.length==1?args[0]:"classes\\Date.bin");
         File f = new File(filename);
         FileInputStream fis = new FileInputStream(f);
         ObjectInputStream ois = new ObjectInputStream(fis);

         // We must read in the same order the object were saved
         String msg = (String) ois.readObject();
         Date today = (Date) ois.readObject();
         ois.close();
        // Print out content
         System.out.println(msg + " " + today.toString());
    } catch (Exception e) {
         e.printStackTrace();
    }
  }
}
```


Input Stream from URL (abstract InputStream)

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

public class InputStreamFromURLApp {
  public static void main(String arg[]){
    try {
      // Create URL reference
      URL url = new URL("http://www.ufp.pt/~rmoreira/LP2/data.txt");
      // Open InputStream from URL
      InputStream is = url.openStream();
      // Read data into byte array buffer
      byte buffer[] = new byte[17];
      is.read(buffer, 0, buffer.length);
      // Print and save data to local file system
      BufferedWriter bw =
           new BufferedWriter(new FileWriter("classes\\data.txt"));
      for (int i = 0; i < buffer.length; i++) {
           System.out.print((char) buffer[i]);
           bw.write((char) buffer[i]);
      }
      is.close();
    } catch (IOException ioe){ ioe.printStacTrace(); }
  }
}
```

Random Access Files

- Open a file for **random access**, i.e., developers may read-from/write-to different positions of the file
- The `RandomAccessFile` class uses a **file pointer** (initially pointing to the beginning of the file – 0 Zero) which stores the file position where we are reading-from or writing-to
- Each read/write operation increments the file pointer by the number of byte transfered
- Developers may move/position the **file pointer** (via **seek()** method) and read bytes from that position onward (or write bytes to that position onward)

Random Output (RandomAccessFile)

```
import java.io.*;

public class RandomFileOutputApp {
  public static void main(String args[]) {
    try {
      // Create file for read-write operations
      RandomAccessFile raf = new RandomAccessFile("Test.txt", "rw");

      // Get current location of file pointer - will print 0 (zero)
      long fp = raf.getFilePointer();
      System.out.println("RandomFileOutputApp-main(): file pointer = "+fp);

      // We could skip the first nbytes of the file:
      //raf.skipBytes(nbytes);
      // or send file pointer to end of file
      raf.seek(raf.length());

      // Subsequent write() operations will be appended to file
      raf.writeUTF("\n new string appended to file");
      raf.writeBytes("\n yet another text line appended");
      raf.close();
    } catch (Exception e){ e.prinStackTrace(); }
  }
}
```

Random Access Files

- The `RandomAccessFile` class works both for **text** and **binary** files but **does not inherit** `InputStream` nor `OutputStream`, therefore, we cannot apply the previously presented stream-oriented filters/wrappers
- However, the `RandomAccessFile` class implements the `DataInput` & `DataOutput` interfaces, hence, it can be used with some **stream filters** (e.g., `DataInputStream`, `DataOutputStream`) and also with specific **random-access** filters/wrappers
- Developers may provide **specific filter/wapper** classes for handling **particular binary/object** files characteristics (e.g., perform *checksum* for input/output error detection)
- However, each **input filter** must know exaclty **what and how** the **output filter** works, i.e., the filters must have **matching** operations (e.g., `read` methods must be coordinated with the `write` methods)

Objects to/from Byte Arrays

- Sometime we may need to **convert objects** to **byte arrays** and vice-versa – we may do so via *Byte Array Streams*:

```
// CONVERT ANY object INTO A byte array
Client c = new Client("Salomé", "Rua de cima");
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);
oos.writeObject(c);
oos.flush();
byte[] bytearray = baos.toByteArray();

// THEN WE MAY CONVERT the byte array BACK TO AN object
ByteArrayInputStream bais = new ByteArrayInputStream(bytearray);
ObjectInputStream ois = new ObjectInputStream(bais);
Client newc = (Client)ois.readObject(); // Do not forget CAST
ois.close();
```

Exercise - ConvertByteArray

```
/** ConvertByteArray: implements generic byte array conversions methods */
public class ConvertByteArray {
    /** toByteArray(): converts any object into a byte array */
    public static byte[] toByteArray(Object obj) throws IOException {
      byte[] ba = null;
      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      ObjectOutputStream oos = new ObjectOutputStream(baos);
      oos.writeObject(obj);
      oos.flush();
      ba = baos.toByteArray();
      oos.close();
      return ba;
    }
    /** fromByteArray(): converts a byte array back to an object */
    public static Object fromByteArray(byte[] ba) throws IOException,
        ClassNotFoundException {
      Object obj = null;
      ByteArrayInputStream bais = new ByteArrayInputStream(ba);
      ObjectInputStream ois = new ObjectInputStream(bais);
      obj = ois.readObject();
      ois.close();
      return obj;
    }
}
```


Some useful java.io classes

- **StreamTokenizer**:

  breaks the contents of a stream into tokens - smallest unit recognized by a text-parsing algorithm (e.g., words, symbols)

  Can be used to parse any text file, e.g., parse a source file into variable names, operators; or parse an HTML file into HTML tags

- **FilenameFilter**:

  used by the list method (in the File class) to determine which files in a directory to list

  Can be used to implement simple regular expression style file search patterns, such as foo*

Some useful java.util.zip classes

- **CheckedInputStream** & **CheckedOutputStream**:
  input and output stream pair that maintains a checksum as the data is being read/written
- **DeflaterOutputStreamand** & **InflaterInputStream**:
  compress or uncompress data as it is being read/written
- **GZIPInputStream** & **GZIPOutputStream**:
  reads and writes compressed data in the GZIP format
- **ZipInputStream** & **ZipOutputStream**:
  reads and writes compressed data in the ZIP format

Example: Produt (1/2)

```
import java.io.*;

public class Product implements Serializable {
    int productCode;
    String productName;
    transient Thread t = new Thread();

    public Product(int code, String name){
        this.productCode = code;
        this.productName = name;
    }

    public void saveToFile(String filename){ // See next slide
    }

    public static void loadFromFile(String filename){ // See next slide
    }

    public String toString() {
        return "Product@[" + productCode + ",  " + productName + "]";
    }
}
```

Example: Produt (2/2)

```
public class Product implements Serializable {
    // Previous code here...
    public void saveToFile(String filename) {
        try {
            File f = new File("Product.bin");
            FileOutputStream fos = new FileOutputStream(f);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(this);
            oos.flush();
            oos.close();
        } catch (Exception e) { e.printStackTrace();}
    }

    public static void loadFromFile(String filename) {
        try {
            File f = new File("Product.bin");
            FileInputStream fis = new FileInputStream(f);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Product p = (Product)ois.readObject();
            ois.close();
            System.out.println(p.toString());
        } catch (Exception e) { e.printStackTrace();}
    }
}
```


Exercise – bank package

- Go back to the **bank package** and make the `Client` and `Account` classes `Serializable`
- Implement the following methods in the `Client` class:
  - `//Saves the client object to a given binary file`
    `public void toObjectFile(String filename):`
  - `//Creates a client object from a given binary file`
    `public static Client fromObjectFile(String filename):`
- Afterwards, use the main method to create several clients associated with several accounts; then save the clients to some file and then read them back and print their information

Exercise: DatabaseApp

```
import java.io.*;

public class ProductDataBaseApp {
  public static void main(String args[]){
    try {
      // Stores & retrieves Product objects using Random Access
      switch (option) {

        case 1 : // Append a new product

        case 2 : // Display a given product

        case 3 : // Update a given product

      }
    } catch (IOException ioe){
        System.out.println("ProductDataBaseApp - main(): "+ioe);
    }
  }
}
```