I/O and serialization

Advanced Programming

Streams

- A **stream** is an abstraction that either **produce or consumes information**

- Streams are implemented in the **java.io** and **java.nio** packages
- Predefined stream: the **java.lang.System** class

I/O streams

- **Byte Streams** I/O of raw binary data.
- **Character Streams** I/O of character data, automatically handling translation to and from the local character set
- **Buffered Streams** optimise input and output by reducing the number of calls to the native API.
- **Scanning and Formatting** allows a program to read and write formatted text
- **I/O from the Command Lin**e describes the Standard Streams and the Console object.
- **Data Streams** handle binary I/O of primitive data type and String values.
- **Object Streams** handle binary I/O of objects.

Byte streams

- Top abstract classes: **InputStream and OutputStream** with methods **read() and write()** that read and write byte streams
- Examples of derived classes:
  - **FileInputStream** – **read()** byte-streams from file
  - **FileOutputStream** – **write()** byte-streams to file
  - **BufferedInputStream** – **read()** byte-streams from current buffer
  - **BufferedOutputStream** – **write()** byte-streams to current buffer
  - **PrintStream -** OutputStream that contains **print() and println()**

Standard Streams

- The Java platform supports the command line interaction in two ways: through the **Standard Streams** and **Console**
- **They are byte streams**

java.lang.System

- Public static final java.lang.System**.in**, java.lang.System**.out**, java.lang.System**.err**
- Contains the following fields:
  - System.in is the keyboard of type **InputStream**
  - System.out is the console of type **PrintStream**
  - System.err of type **PrintStream**

The Console class

- A more advanced alternative to the **Standard Streams** is the Console
  - To read and write from a console if one exists
  - Console has no constructor
  - It is final
- One can get a Console object by invoking the method public Console console()

Console c = System.console();

The Console class

- Besides the usual methods to read from a line, Console has the method

char[] readPassword()

- that reads a string entered in the keyboard until the user presses ENTER

readPassword()

- This method helps secure password entry in two ways.
  - First, it suppresses echoing, so the password is not visible on the user's screen.
  - Second, it returns a **character array, not a String (it is immutable!)**, so the password **can be overwritten**, removing it from memory as soon as it is no longer needed

Have you tried?

- Console does not work in an IDE!

- see file BBReader.java

[1]http://unicode-table.com/en/

Character streams

- Character streams use UNICODE[1] so that they can be internationalised
- Top abstract classes: **Reader** and **Writer** with methods read() and write() that read or write character streams
  - Examples of derived classes:

BufferedReader *– characters stream*
StringReader *– reads form string*
BufferedWriter *– characters stream*
StringWriter *– writes to string*
FileReader *– reads from file*
FileWriter *– writes to file*

InputStreamReader *- translates bytes to characters*
OutputStreamWriter *- translates characters to bytes*
LineNumberReader *– counts lines*
PrintWriter *– contains print() and println()*

int read()

```
Reader reader = new FileReader("c:\data\myfile.txt”);
   int data = reader.read();
   while(data != -1){
      char dataChar = (char) data;
      data = reader.read();
   }
```

InputStream returns one byte at a time, i.e. a value between 0 and 255 (or -1 if the stream has no more data)

Reader returns a char at a time, i.e. a value between 0 and 65535 (or -1 if the stream has no more data)

“-1” tells there are no more data

Wrapping byte into char streams

- Standard Streams for historical reasons are byte streams
- **System.out and System.err** are defined as internal objects of System of type **PrintStream**
- OK! **PrintStream is byte stream** but uses an internal character stream object to **emulate** many of the features of **character streams**

Wrapping byte into char streams

- **System.in** is a byte stream with no character stream features.
- To use **System.in** as a character stream, **wrap** System.in in InputStreamReader

```
InputStreamReader wrin = new InputStreamReader(System.in);
```

Reading characters / strings

- BufferedReader
- **int read() throws IOException** returns integer value for a character (needs to be casted to char), -1 if it is EOL (end of line)
- **String readLine() throws IOException** returns String
- BufferedReader store characters in a **buffer** for further use

Unbuffered I/O

- Each read or write request is handled directly by the underlying OS
- This can make a program much less efficient:
  - each request often triggers **disk access or network activity**
- E.g.: Inputstream / Outputstream

Buffered I/O

- Buffered input streams read data from a memory area known as a buffer
- Buffered output streams write data to a buffer
- Buffers are automatically emptied, but one can use flush() to force it
- A program can convert an unbuffered stream into a buffered stream using the wrapping idiom

Reading from console input

- Wrapping System.in as object of BufferedReader by passing an anonymous object of type InputStreamReader with parameter System.in

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

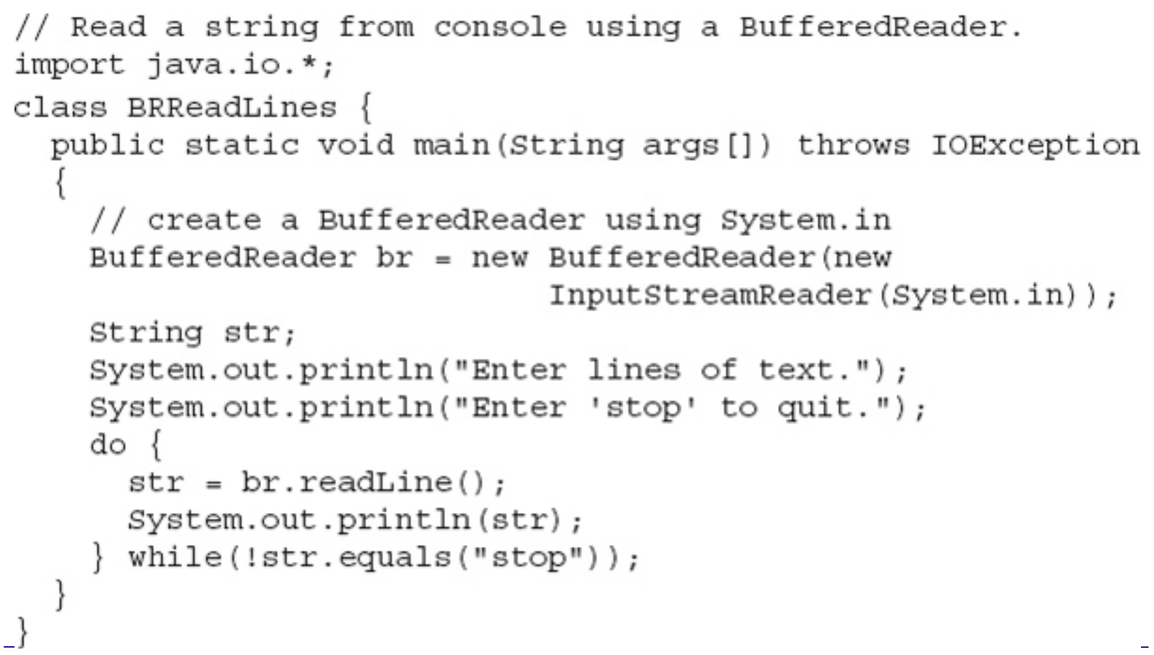
- With constructors

BufferedReader(Reader myReader)

InputStreamReader(InputStream myInputStream)

It takes a byte inputstream, it translates it to a character stream by wrapping it into InputStreamReader and then store it in a buffer for further use by wrapping it into a BufferedReader

Example from reference book



```
// Read a string from console using a BufferedReader.
import java.io.*;
class BRReadLines {
  public static void main(String args[]) throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new
                            InputStreamReader(System.in));
    String str;
    System.out.println("Enter lines of text.");
    System.out.println("Enter 'stop' to quit.");
    do {
      str = br.readLine();
      System.out.println(str);
    } while(!str.equals("stop"));
  }
}
```

Flushing

- Streams are often accessed by threads that periodically empty their content and, for example, display it on the screen, send it to a socket (socket is an endpoint for communication between two machines) or write it to a file
- This is done for performance reasons
- Flushing an output stream means that we want to stop, wait for the content of the stream to be completely transferred to its destination, and then resume execution with the stream empty and the content sent.

flush()

- **flush():** the streams are stored in a temporary memory location in our computer called **buffer**
- When all buffers are full then invoking flush() empty them and write out the streams on the device we want (i.e. in a file or console)
- **close()** invokes flush() by default, but sometimes we want to force flush() even before we close the stream for performance reasons

Using PrintWriter for console output

```
// Demonstrate PrintWriter
import java.io.*;

public class PrintWriterDemo {
  public static void main(String args[]) {
    PrintWriter pw = new PrintWriter(System.out, true);

    pw.println("This is a string");
    int i = -7;
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

PrintWriter class

- It is one of character based classes so it is better than using System.out for internationalization
- If the return value is not primitive, its print methods invoke toString() of Object to display the results

PrintWriter(OutputStream outputStream, boolean flushingOn)

the boolean parameter indicates whether **flush()** is performed and print() and println() empty the buffer automatically

Reading/writing/manipulating **files**

- **Classes** in java.io:
  - FileInputStream , FileOutputStream,
  - File,
  - FileReader, FileWriter, PrintWriter,
- **Methods**: read(), readLine(), print(), println(), flush(), write(), close() a file
- **Exceptions**: IOException (FileNotFoundException), IOError, SecurityException (attempt to open a file with security policy, better used with applet that have default security manager)

FileInputStream constructors

- FileInputStream(String filePath)
- FileInputStream(File fileObj)

FileInputStream fileStream=new FileInputStream("/ game.bat");

or

File file = new File("/game.bat");

FileInputStream afileStream = new FileInputStream(file)

FileOutputStream constructors

- FileOutputStream(String filePath)
- FileOutputStream(File fileObj)
- FileOutputStream(String filePath, boolean append)
- FileOutputStream(File fileObj, boolean append)
  - filePath is the full path
  - boolean determines whether to append a content of on the file

close()

- **void close() throws IOException** – you can use it in the body of a method, in a finally block or you do not use it with **try-with-resources()** block(after SDK 1.7)
  - Not closing a file results in memory leaks that is allocating memory to unused resources
- Exercise: instantiate a file as you wish and try to close it outside the method block or in a finally block etc…

Writing files as chars or bytes

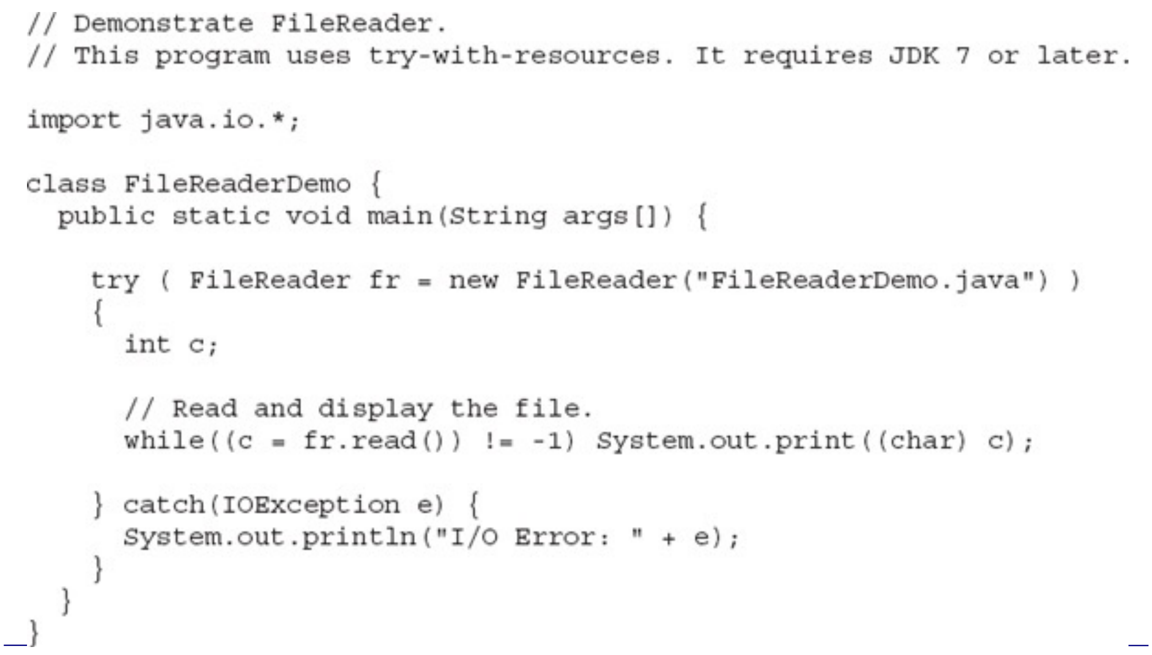
```
PrintWriter writer = new PrintWriter("the-file-name.txt", "UTF-8");
writer.println("The first line");
writer.close();

byte dataToWrite[] = //...
FileOutputStream out = new FileOutputStream("the-file-name.txt");
out.write(dataToWrite);
out.close();
```

see file B_ReadLines.java and CopyFile.java

FileReader

- It reads the content of a file
- Constructors:
  - FileReader(String filePath)
  - FileReader(File fileObj)

Example from the book



```
// Demonstrate FileReader.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

    try ( FileReader fr = new FileReader("FileReaderDemo.java") )
    {
      int c;

      // Read and display the file.
      while((c = fr.read()) != -1) System.out.print((char) c);

    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

FileWriter

- FileWriter creates a Writer that you can use to write to a file
- FileWriter will create the file before opening it for output when you create the object.
- In the case where you attempt to open a read-only file, an **IOException** will be thrown.

FileWriter

- Constructors
  - FileWriter(String filePath)
  - FileWriter(String filePath, boolean append)
  - FileWriter(File fileObj)
  - FileWriter(File fileObj, boolean append)

The File class

- An abstract representation of file and directory pathnames.
- It can store a file or a directory. In the last case, use **isDirectory()** to check it. Only if it returns TRUE you can use the **list()** method that returns the String array of all the files in a directory
- Sometimes you have sub-directories as in the following example
- see class MyFolder.java and FileDemo.java

Reading a directory

```
// Using directories.
import java.io.File;

class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
        }
      }
    } else {
      System.out.println(dirname + " is not a directory");
    }
  }
}
```

Here the sub-directories are listed as files of one directory

Here I can see the difference between a sub-directory and a file

Constructors

- File(String directoryPath)
- File(String directoryPath, String filename)
- File(File dir, String filename)
- File(URI uri)

File f1 = File(“/”);

File f2 = File(“/”, “game.bat”);

File f3 = File(f1, “game.bat”);

```
// Demonstrate File.
import java.io.File;

class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File("/java/COPYRIGHT");

    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "might be a named pipe");
    p(f1.isAbsolute() ? "is absolute" : "is not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

Exercise

- Count the number of projects of your workspace and their total size

Note

- To use try-with-resources classes (resources) must implement the closeable interface
- The class File does not do it by default
- The stream abstract classes (byte or char) yes

Limiting the number of files

- To limit the number of files returned from list() we can use the interface FilenameFilter
- It allows to filter the files by matching a string

  accept(File thisDirectory, String matchName)

  Directory in which we look up, string we want to match

- and then use

  String[] list(FilenameFiler myfiles)

Example

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
  String ext;

  public OnlyExt(String ext) {
    this.ext = "." + ext;
  }

  public boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}
```

Drawbacks of File

- Many methods do not throw exceptions when they failed (depending on the access defined by the OS)
- Instances of File are immutable: the abstract path name represented by a File object will never change (see code example “RenamedFile.java”

Drawbacks of File

- The rename() method didn't work consistently across OSs
- **Accessing file metadata is inefficient** (file permissions, file owner, and other security attributes)

Drawbacks of File

- **Many of the File methods didn't scale**. Requesting a large directory listing over a server could stack.
- Large directories could also cause memory leak, resulting in a denial of service

The java.nio package

- The NIO system is built on two foundational items:
  - A *buffer* holds data
  - A *channel* represents an open connection to an I/O device, such as a file or a socket

  - We will review two major **Files** and **Paths** classes

java.nio.file.Path (I)

- A Java Path instance represents a path in the file system
- A path can point to either a file or a directory
    - A path can be absolute or relative

java.nio.file.Paths (C)

- To get an object of Path (is an interface!) we use factory methods (often static calls)

```
Path pathOne = Paths.get(“data/myfile.txt");

or Path pathOne = Paths.get(“c:\\data\\myfile.txt");

Path pathTwo = FileSystems.getDefault().getPath("data", “myfile.txt");


BufferedReader reader = Files.newBufferedReader(pathOne,
StandardCharsets.UTF_8);
```

Relative path

- base path : “data”
- relative path: “projects/myFoo.txt”

```
Path pathBase = Paths.get(“data");
Path pathFolder = Paths.get(“data”, “projects”);
Path pathFile = Paths.get(“data”, “projects/myFoo.txt”);

Path currentDir = Paths.get(".");
```

java.nio.file.Files (C)

- The java.nio.file.Files class works with java.nio.file.Path instances

```
boolean pathExists =
     Files.exists(pathOne,new LinkOption[]{…});
```

- Files.createDirectory() method creates a new directory from a Path instance

```
try {
    Path newDir = Files.createDirectory(pathTwo);
} catch(FileAlreadyExistsException e){
    // the directory already exists.
} catch (IOException e) {
    //something else went wrong
    e.printStackTrace();
}
```

Serialisation

- When group of objects or states can be transmitted as one entity and then at arrival reconstructed into the original distinct objects

Serialisation

- Serialisation is the process to write the state of an object to a byte stream:
  - An object is represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

Serialisation

- Later one may restore these objects by using the deserialisation process:
  - it can be read from the persistent storage (e.g., a file, or a socket) and deserialised it that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Top classes

- Top classes **ObjectInputStream** and **ObjectOutputStream**

Java object serialisation

- The `ObjectOutputStream` class contains the method

```
public final void writeObject(Object x)
throws IOException
```

- The method **serialises** an Object and sends it to the output stream

Java object deserialisation

- Similarly, the **ObjectInputStream** class contains the method for **deserialising** an object:

```
public final Object readObject() throws
IOException, ClassNotFoundException
```

- This method retrieves the next Object out of the stream and deserialises it
- Casting to appropriate data type is needed as the method returns Object

How to build a serialisable class

- First a class must implement the interface java.io.Serialisable
- **All of the class fields** must be **serializable** too
- If a field is not serializable, it must be marked as **transient**
- see file Employee.java

Deserialisation

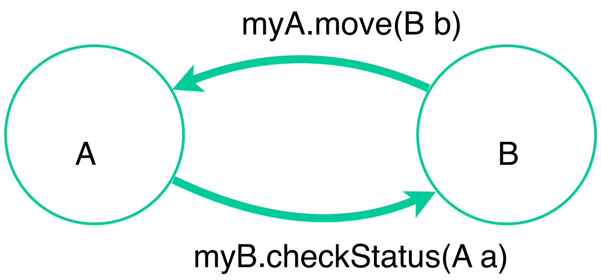
- Read an object from the ObjectInputStream.
- The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its super-types are read.
- Objects referenced by this object are read transitively so that a complete equivalent graph of objects is reconstructed by readObject.

- The root object is completely restored when all of its fields and the objects it references are completely restored.

Directed graph of object relations

- Assume that an object to be serialised has references to other objects, which, in turn, have references to other objects
- This set of objects and the relationships among them form a directed graph

Circular reference



Circular references

- There may also be circular references within this object graph.
  - That is, object A may contain a reference to object B, and object B may contain a reference back to object A
- **Not a good design**
- Objects may also contain references to themselves and self references

```
class A{
   B b1;
   A( ) {b1=new B();}
}

class B{
   A a1;
   B( ){a1=new A();}
}

public class Demo{
   public static void main(String[] args){
      A obj=new A();
// throw away the reference from the stack
      obj=null;   // inaccessible circular
references now exists
   }
}
```

Stack overflow: not a circular reference

```
class A{
   private B b;
   public void setB(B b) { this.b = b;}
}

class B {
   private A a;
   public void setA(A a) { this.a = a;}
}

public class Demo{
   public static void main(String[] args) {
      A myA = new A();
      B myB = new B();
      // Make the objects refer to each other
(creates a circular reference)
      myA.setB(myB);
      myB.setA(myA);
/* Throw away the references from the main
*method; the two objects are still referring to
/*each other
      myA = null;
      myB = null;
   }
}
```

Circular reference

Exercise

- draw the stack and heap models for the above code

Serialize the graph

- When we serialise an object at the top of an object graph, all of the other referenced objects are recursively located and serialised
- Similarly, during the process of deserialisation, all of these objects and their references are correctly restored

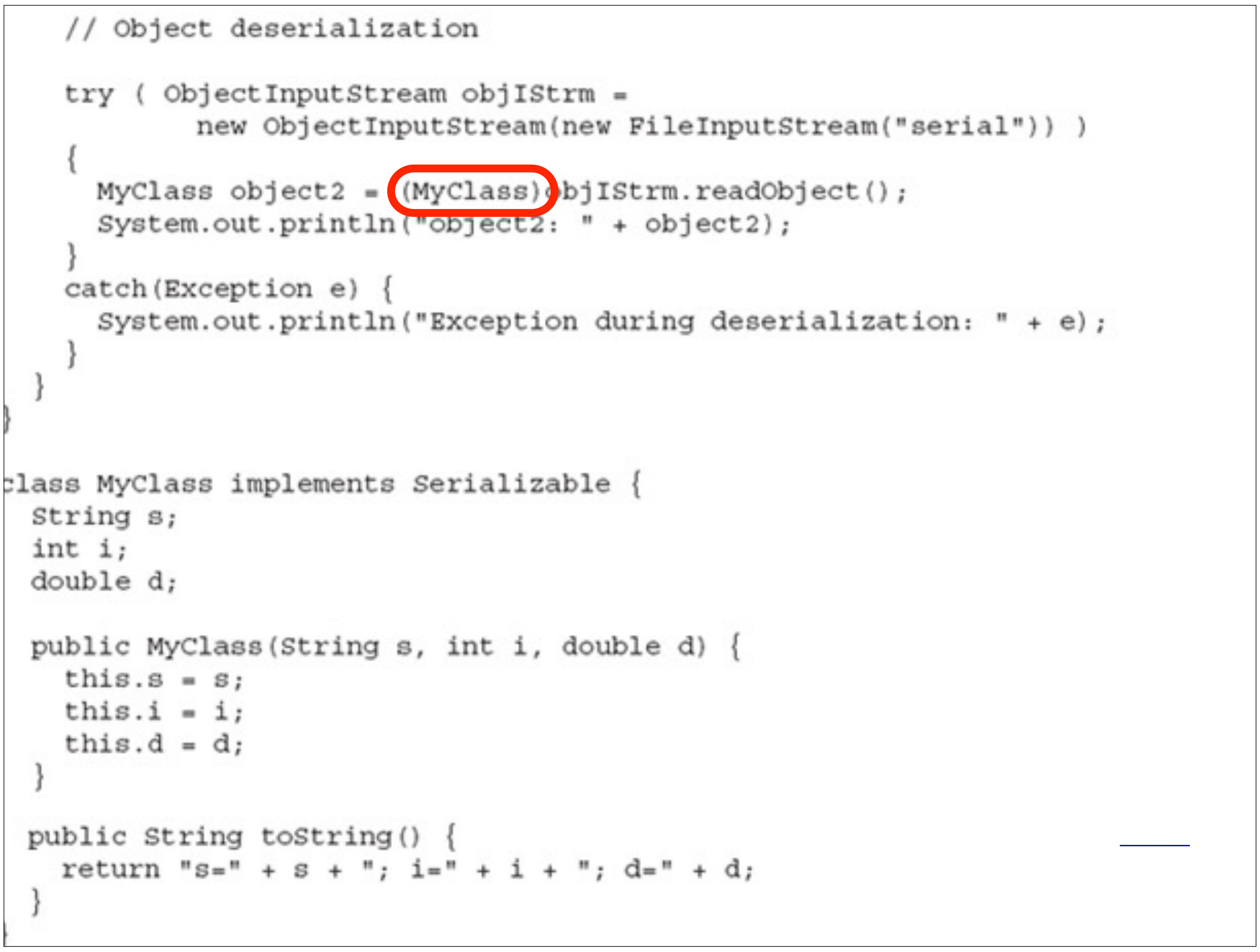
Properties of serialization

- The entire process is JVM independent, meaning an object can be serialised on one platform and deserialised on an entirely different platform
- Useful with **Remote Method Invocation** (RMI)
  - RMI allows object of one machine to call methods of objects in another machine
    - An object can be passed of parameter of the remote method. The sending machine serialize it and transmit it, while the receiving machine deserialize it

```
import java.io.*;

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

    // Object serialization

    try ( ObjectOutputStream objOStrm =
            new ObjectOutputStream(new FileOutputStream("serial")) )
    {
      MyClass object1 = new MyClass("Hello", -7, 2.7e10);
      System.out.println("object1: " + object1);

      objOStrm.writeObject(object1);
    }
    catch(IOException e) {
      System.out.println("Exception during serialization: " + e);
    }
```

- A **FileOutputStream** is created that refers to a file named "serial", and an **ObjectOutputStream** is created for that file stream.
- invoking **writeObject( )** of the class ObjectOutputStream to serialize the object of FileOutputStream.
- The object of FileOutputStream is flushed and closed.

- ObjectOutputStream is used to create binary representation of Serializable objects



```
   // Object deserialization

   try ( ObjectInputStream objIStrm =
            new ObjectInputStream(new FileInputStream("serial")) )
   {
     MyClass object2 = (MyClass)objIStrm.readObject();
     System.out.println("object2: " + object2);
   }
   catch(Exception e) {
     System.out.println("Exception during deserialization: " + e);
   }
 }
}

class MyClass implements Serializable {
  String s;
  int i;
  double d;

  public MyClass(String s, int i, double d) {
    this.s = s;
    this.i = i;
    this.d = d;
  }

  public String toString() {
    return "s=" + s + "; i=" + i + "; d=" + d;
  }
}
```

- Note that MyClass is defined to implement the Serializable interface.
- If this is not done, a NotSerializableException is thrown.