Basic Concepts - Streams, Files, Etc.

The Java Model for I/O
Streams - What, why and interaction with file systems.
Files - What, why (history) and interaction with programming languages.
The Java class hierarchy for Stream I/O
- **Streams of bytes**
- **Streams of Java primitive types**
- **Streams of characters**
- **Streams of object instances**

Serializability
Text file IO
Random Access IO

Streams in Java

1. A stream is a flow or sequence of typed units of data. There are streams of ints, streams of chars, streams of Strings, streams etc. There are, therefore many kinds of streams.

2. Streams have direction: input and output.

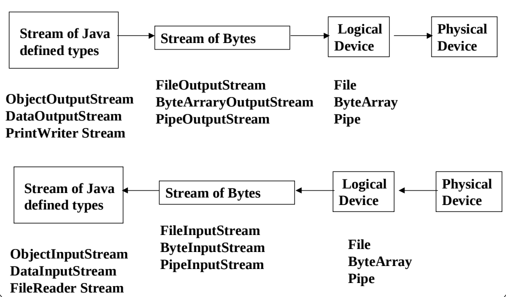
3. Streams have sources and targets.
Sources for output are values of variables generated in programs.
Targets for outputs include the system console of your computer, a file implemented by an operating system, a buffer in process memory or a network connection to another computer.

The sources for input streams are keyboards, mouses, files and network connections.

4. Streams are implemented as Java classes and must be declared and instantiated.

5. System.out and System.in are special streams provided for writing to and reading from the console screen of the host computer.

Logical Structure of File I/O In Java



The prefix “File” in front of an IO class name means that the operations are on streams of bytes.
(FileInputStream) - binary file

The prefix “Data” in front of an IO class name means that the operations are on streams of Java primitive types, ints, doubles, etc..
(DataInputStream) - binary file

The prefix “Object” in front of an IO class name means that the operations are on streams of instances of user defined classes.
(objectInputStream) - binary file

Appearance of the words “Reader” or “Writer” in the name of an IO class means that this class reads or writes streams of characters.
(FileReader) - text file

Appearance of the word “Buffered” in the names of an IO class means that this class “buffers” data transfers.
(BufferedInputStream)

Why Four Types of IO In Java?

Application Requirements

- **Some applications need to access all the data in a file.**
  Streams
- **Some applications need to access only selected portions of a file.**
  Random Access
- **Some applications need control over what is written and read down to the level of primitive type instances.**
  DataOutput and DataInput Streams
- **Some applications can take advantage of the convenience of writing and reading entire objects.**
  ObjectOutput and ObjectInput Streams
- **Java programs must read files written by programs written in other language systems and write files which can be read by programs written in other languages..**
  PrintWriter and PrintReader Streams

Why the implementation is the way it is.

1. Operating systems all support the concept of a file as a stream of <u>bytes</u>.
- **Java targets transparent execution across multiple platforms.**
- **So other representations must be mapped to and from streams of bytes.**

2. Some applications need to access all of the data of a file and reading sequentially down the stream of bytes is the most efficient form of access.
- **Therefore streams are natural.**

3. Some applications need to look only at selected portions of a file and most operating system provide a capability for positioning the location at which the next read or write occurs as a position in the stream of bytes.
- **Random access is implemented by operations which specify positions in a stream of bytes. (a file)**

Text, Binary and Object Formats for Data Storage and Communication

Text files - are written in formats which are agreed upon across processors and language systems. A file written as a text file in Java can be read by programs written in other language systems provided that these programs follow the standard conventions for writing and reading text files.
Text files are streams of characters.

Binary files - Are written in formats which Java understands but which are not necessarily readable by any program not written in Java.
Contain streams of instances of the primitive data types of Java.
Binary files are written using the binary representation of the primitive data types being written.

Object files - Are written in formats specific to Java objects.
Contain streams of Java objects.
Maintain relationships among object instances. (Serializability)

Files and File Abstractions

Files are abstractions defined for persistent storage and access of data of arbitrary types. Most applications require persistent storage of data and/or programs.

Files are either:
Sequential - Streams of entities (bits, bytes, ints, chars, objects, etc.)
Random - Addressable blocks of (bits, bytes, etc.)

Files are defined by operating systems. The file types provided by an operating system are used by all of the language systems implemented on a given computer system.

Files may be represented explicitly in a programming language or a programming language may choose to implement its own set of abstractions and map them to the files provided by the host OS when persistent storage is required. Java chooses the second approach.

How did current file abstractions develop?

1. Memory Systems
   Executable and transient memory.
      Transistor based - turn off power and memory content is lost.
      Each cell is “equally” accessible.
   Non-executable but persistent memory.
      Magnetic spot based (or reflective spot based) - reasonably persistent.
2. Early form of persistent memory was magnetic tape which is a serially accessible memory. (Or printers which are write only serial devices.)

3. The logical abstraction for storage of data on a magnetic tape was a sequence of entities. The logical operations on a sequence of entities are:
      read the “next” k entities on the sequence of entities.
      write k entities to the “next” k positions to extend the sequence.
   “Next” implies some notion of position.
   Or a file looks a lot like a one-way linked list where the entries are of size k entities of some type.

4. But nowadays files are stored on disks which are directly addressable by blocks of size k.
   And many applications can benefit from being able to access data by its content and content can be mapped to position in an index space.
   So we have the concept of “random access” files where the file consists of a set of blocks of k entities.
   The operating system provides access by block address.
   The application and the operating system together provide mapping from the content of the blocks the application defines and the positions on the disk (or disks) of the blocks of data.

5. Abstractions for persistent storage were typically provided by libraries of routines which were device specific. Print , Write Tape, etc.

6. Nowadays applications on one system often interact with other applications on different systems. That is, applications communicate over networks. This new development has often been integrated into some operating systems by abstracting the network connection to look like a file device.

7. The problem which must be solved has now been generalized to include interactions among applications. The solution has been to extend the abstractions used for implementation of persistent storage.

8. Operating systems and languages languages have began to provide less device specific abstractions. Streams as defined in Java are an example of these device “independent” abstractions.

Files, File Names and Path Names

There is a class File provided in the java.io package which enables you to create files while executing in Java Programs. It also enables you to determine the properties of files such as whether or not a file of a given name exists and the length of a file in bytes.
File names for constructors of stream classes are just instances of type String.
But these file names are passed on to the host OS by the Java runtime system so must have meaning to the host OS.
Path names are the names OS’s use to locate files in a tree structure so path names are host OS dependent.
If the string name you give as a constructor to a stream class has no path then the system assumes you are reading from and/or writing to files defined in the current directory.
System.getProperty(“user.dir”) - Will get current directory.

java.io Class File

java.lang.Object
  |
  +--java.io.File

public class File extends Object implements Serializable, Comparable

Class File implements an abstract representation of file and directory pathnames.

User interfaces and operating systems use system-dependent pathname strings to name files and directories. This class presents an abstract, system-independent view of hierarchical pathnames. An abstract pathname has two components:

1. An optional system-dependent prefix string, such as a disk-drive specifier, "/" for the UNIX root directory, or "\\" for a Win32 UNC pathname, and

2. A sequence of zero or more string names.

Each name in an abstract pathname except for the last denotes a directory; the last name may denote either a directory or a file. The empty abstract pathname has no prefix and an empty name sequence.

The conversion of a pathname string to or from an abstract pathname is inherently system-dependent. When an abstract pathname is converted into a pathname string, each name is separated from the next by a single copy of the default separator character. The default name-separator character is defined by the system property file.separator, and is made available in the public static fields separator and separatorChar of this class. When a pathname string is converted into an abstract pathname, the names within it may be separated by the default name-separator character or by any other name-separator character that is supported by the underlying system.

A pathname, whether abstract or in string form, may be either absolute or relative. An absolute pathname is complete in that no other information is required in order to locate the file that it denotes. A relative pathname, in contrast, must be interpreted in terms of information taken from some other pathname.

By default the classes in the java.io package always resolve relative pathnames against the current user directory. This directory is named by the system property user.dir, and is typically the directory in which the Java virtual machine was invoked.

For our purposes we will assume that the files used in a program are in the directory in which the Java VM was invoked.

There are many methods, too many to list or discuss.
See http://java.sun.com/products/jdk/1.2/docs/api/java/io/File.html

Constructor Summary

File(File parent, String child)
    Creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)
    Creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)
    Creates a new File instance from a parent pathname string and a child pathname string.

Make Directory Method

boolean mkdir()
    Creates the directory named by this abstract pathname.

Method Summary

String[] list()
    Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
String getPath()
    Converts this abstract pathname into a pathname string.
boolean isDirectory()
    Tests whether the file denoted by this abstract pathname is a directory.
String getCanonicalPath()
    Returns the canonical pathname string of this abstract pathname.

There are numerous other methods.

InputStream - An abstract class defining methods for input of bytes.

FileInputStream - A class implementing methods for input of streams of bytes and arrays of bytes.

OutputStream - An abstract class defining methods for output of bytes.

FileOutputStream - A class implementing methods for output of streams of bytes and arrays of bytes.

```
import java.io.*;
public class WriteMatrix1 {
  static double[][] data = {
    { Math.exp(2.0), Math.exp(3.0), Math.exp(4.0) },
    { Math.exp(-2.0), Math.exp(-3.0), Math.exp(-4.0) },
  };
  public static void main(String[] args) {
    int row = data.length;
    int col = data[0].length;
    int i, j;
    for (i = 0; i < row; i++) {
      for (j = 0; j < col; j++) {
           System.out.println("data[" + i + "][" + j + "] = " + data[i][j]);
      }
    }
```



```
if (args.length > 0) {
   try {
        FileOutputStream out = new FileOutputStream(args[0]);
        writeInt(row, out);
        writeInt(col, out);
        for (i = 0; i < row; i++) {
          for (j = 0; j < col; j++) {
            writeDouble(data[i][j], out);
          }
        }
        out.close();
   } catch (IOException e) {}
  }
 }
```

FileOutputStream out takes a stream of bytes and writes it to the file named by args[0]. The writeDouble method explicitly converts doubles to bytes.


```
public static void writeInt(int i, OutputStream out)
      throws IOException {
    byte[] buf = new byte[4];
    for (int k = 3; k >= 0; k--) {
      buf[k] = (byte)(i & 0xFF);
      i >>>= 8;
    }
    out.write(buf);
  }
  public static void writeDouble(double d, OutputStream out)
      throws IOException {
    byte[] buf = new byte[8];
    long l = Double.doubleToLongBits(d);
    for (int k = 7; k >= 0; k--) {
      buf[k] = (byte)(l & 0xFF);
      l >>>= 8;
    }
    out.write(buf);
  }
}
```

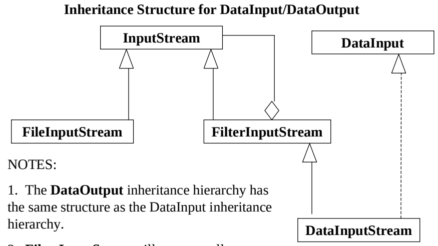
These methods must be written by the user when byte IO is used.


```
import java.io.*;
public class ReadMatrix1 {
  static double[][] data;
  public static void main(String[] args) {
    if (args.length > 0) {
      try {
        FileInputStream in = new FileInputStream(args[0]);
        int row = readInt(in);
        System.out.println("row = " + row);
        int col = readInt(in);
        System.out.println("col = " + col);
        data = new double[row][col];
        for (int i = 0; i < row; i++) {
          for (int j = 0; j < col; j++) {
            data[i][j] = readDouble(in);
            System.out.println("data[" + i + "][" + j + "] = " + data[i][j]);
          }
        }
      } catch (IOException e) {}
    }
  }
```



```
public static int readInt(InputStream in)
      throws IOException {
    byte[] buf = new byte[4];
    in.read(buf);
    int i = 0;
    for (int k = 0; k < 4; k++) {
      i <<= 8;
      i += (((int) buf[k]) & 0xFF);
    }
    return i;
  }
  public static double readDouble(InputStream in)
      throws IOException {
    byte[] buf = new byte[8];
    in.read(buf);
    long l = 0;
    for (int k = 0; k < 8; k++) {
      l <<= 8;
      l += (((int) buf[k]) & 0xFF);
    }
    return Double.longBitsToDouble(l);
  }
}
```


Inheritance Structure for DataInput/DataOutput



NOTES:

1. The **DataOutput** inheritance hierarchy has the same structure as the DataInput inheritance hierarchy.

2. **FilterInputSteam** will not normally appear in program text. It defines and implements methods for reading bytes and converting from bytes to other types. **DataInputStream** is a filter which converts bytes to Java primitive data types.

DataInput is an interface defining the operations for input of primitive Java types.

public interface DataInput
The DataInput interface provides for reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types. There is also a facility for reconstructing a String from data in Java modified UTF-8 format.
It is generally true of all the reading routines in this interface that if end of file is reached before the desired number of bytes has been read, an EOFException (which is a kind of IOException) is thrown. If any byte cannot be read for any reason other than end of file, an IOException other than EOFException is thrown. In particular, an IOException may be thrown if the input stream has been closed.

Method Summary

boolean readBoolean()
Reads one input byte and returns true if that byte is nonzero, false if that byte is zero.
byte readByte()
Reads and returns one input byte.
char readChar()
Reads an input char and returns the char value.
double readDouble()
Reads eight input bytes and returns a double value.


```
import java.io.*;
public class WriteMatrix2 {
  static double[][] data = {
    { Math.exp(2.0), Math.exp(3.0), Math.exp(4.0) },
    { Math.exp(-2.0), Math.exp(-3.0), Math.exp(-4.0) },
  };
  public static void main(String[] args) {
    int row = data.length;
    int col = data[0].length;
    int i, j;
    for (i = 0; i < row; i++) {
      for (j = 0; j < col; j++) {
            System.out.println("data[" + i + "][" + j + "] = " + data[i][j]);
      }
    }
```



```
if (args.length > 0) {
    try {
        DataOutputStream out =
            new DataOutputStream(new FileOutputStream(args[0]));
        out.writeInt(row);
        out.writeInt(col);
        for (i = 0; i < row; i++) {
          for (j = 0; j < col; j++) {
            out.writeDouble(data[i][j]);
          }
        }
        out.close();
    } catch (IOException e) {}
  }
 }
}
```

DataOutputStream out takes a streams of Java primitive type instances and converts them to a stream of bytes and writes them to the instance of FileOutputStream which is the argument of its constructor

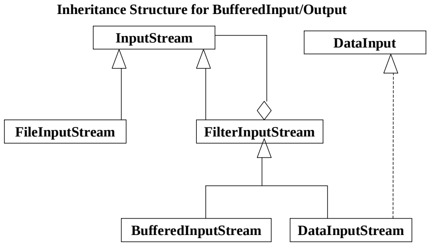
FileOutputStream(args[0] writes out a stream of bytes to the file of name defined by the value of args[0].

writeDouble(data[i][j]) is a method of DataOutputStream which takes the double value in data[i][j] and converts it to eight bytes and writes them to FileOutputStream.


```
import java.io.*;
public class ReadMatrix2 {
  static double[][] data;
  public static void main(String[] args) {
    if (args.length > 0) {
      try {
            DataInputStream in =
              new DataInputStream(new FileInputStream(args[0]));
            int row = in.readInt();
            System.out.println("row = " + row);
            int col = in.readInt();
            System.out.println("col = " + col);
            data = new double[row][col];
            for (int i = 0; i < row; i++) {
              for (int j = 0; j < col; j++) {
                data[i][j] = in.readDouble();
                System.out.println("data[" + i + "][" + j + "] = " + data[i][j]);
              }
            }
      } catch (IOException e) {}
    }
  }
}
```

The read request initiated in DataInputStream in is passed to the instance of FileInputStream which reads in the bytes and passes them back to in.

Inheritance Structure for BufferedInput/Output



A BufferedInputStream "buffers" data from the file in advance of the in-memory operations. For example, a BufferedInputStream will read a line of text for subsequent extraction of the words of the line of text.

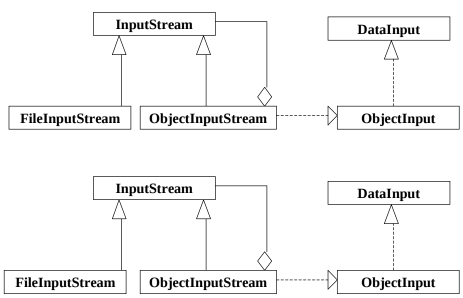

```
import java.io.*;
public class WriteMatrix3 {
static double[][] data = {
    { Math.exp(2.0), Math.exp(3.0), Math.exp(4.0) },
    { Math.exp(-2.0), Math.exp(-3.0), Math.exp(-4.0) },
  };
  public static void main(String[] args) {
    int row = data.length;
    int col = data[0].length;
    int i, j;
    for (i = 0; i < row; i++) {
      for (j = 0; j < col; j++) {
            System.out.println("data[" + i + "][" + j + "] = " + data[i][j]);
      }
    }
```

```
if (args.length > 0) {
    try {
        DataOutputStream out =
          new DataOutputStream(new BufferedOutputStream(new
                        FileOutputStream(args[0])));
        out.writeInt(row);
        out.writeInt(col);
        for (i = 0; i < row; i++) {
          for (j = 0; j < col; j++) {
            out.writeDouble(data[i][j]);
          }
        }
        out.close();
    } catch (IOException e) {}
  }
 }
}
```

DataOutputStream converts instances of Java primitive types into a stream of bytes and writes them to a BufferedOutputStream which in turns writes the bytes to a FileOutputStream which writes the bytes to the file with name specified by args[0].


```
import java.io.*;
public class ReadMatrix3 {
  static double[][] data;
  public static void main(String[] args) {
    if (args.length > 0) {
      try {
           DataInputStream in =
             new DataInputStream(new BufferedInputStream(new
                          FileInputStream(args[0])));
           int row = in.readInt();
           System.out.println("row = " + row);
           int col = in.readInt();
           System.out.println("col = " + col);
           data = new double[row][col];
           for (int i = 0; i < row; i++) {
             for (int j = 0; j < col; j++) {
               data[i][j] = in.readDouble();
               System.out.println("data[" + i + "][" + j + "] = " + data[i][j]);
             }
           }
      } catch (IOException e) {}
    }
  }
```


Inheritance Hierarchy for Object/IO



public interface **ObjectOutput** extends DataOutput

ObjectOutput extends the DataOutput interface to include writing of objects. DataOutput includes methods for output of primitive types, ObjectOutput extends that interface to include objects, arrays, and Strings.

void **writeObject**(Object obj)

Write an object to the underlying storage or stream.

public interface **ObjectInput** extends DataInput

ObjectInput extends the DataInput interface to include the reading of objects. DataInput includes methods for the input of primitive types, ObjectInput extends that interface to include objects, arrays, and Strings.

Object **readObject**()

Read and return an object.

Serializing and Deserializing of Objects

There are two cases of Serialization:

1. For classes where all instance variables are primitive Java types.

Each instance variable is converted to a representation in bytes. The object is assigned a "serial number" and written out as an array of bytes in the order in which the instance variables are declared in the class specification.

2. Serialization for classes where one or more instance variables are instances of user defined classes is more complex.

Then the system attempts to insure that each instance of a class is written out only once.

An instance which is an instance variable in another other is represented in the storage of the containing instance by its serial number.

3. The example in Jia illustrates only the simple case.


```
import java.io.*;
public class WriteMatrix5 {
  static double[][] data = {
    { Math.exp(2.0), Math.exp(3.0), Math.exp(4.0) },
    { Math.exp(-2.0), Math.exp(-3.0), Math.exp(-4.0) },
  };
  public static void main(String[] args) {
    int row = data.length;
    int col = data[0].length;
    int i, j;
    for (i = 0; i < row; i++) {
      for (j = 0; j < col; j++) {
            System.out.println("data[" + i + "][" + j + "] = " + data[i][j]);
      }
    }
```

```
if (args.length > 0) {
    try {
        ObjectOutputStream out =
            new ObjectOutputStream(new FileOutputStream(args[0]));
        out.writeObject(data);
        out.close();
    } catch (IOException e) {}
  }
 }
}
```

The **ObjectOutputStream** writes a stream of byte arrays to the **FileOutputStream** which writes them to the file named by **args[0].**

out.writeObject(data) converts the array of doubles to an array of bytes which is passed to the **FileOutputStream.**

Serialization when classes have instances of other classes as instance variables.

1. **Layout of object instances in storage.**
2. **Desirable properties for a serialization mechanism.**
3. **The Java serialization mechanism.**
4. **Some examples of serialization.**

An example program to demonstrate memory layout for object instances.

```
class ObjectFileTest
{  public static void main(String[] args)
        { Employee[] staff = new Employee[3];
         Employee harry = new Employee("Harry Hacker",
              35000, new Day(1989,10,1));
        Manager carl = new Manager("Carl Cracker",
              75000, new Day(1987,12,15));
        Manager tony = new Employee("Tony Tester",
              38000, new Day(1990,3,15));
         Staff[0] = harry;
         Staff[1] = carl;
         Staff[2] = tony;
         int i;
         for (i = 0; i < Staff.length; i++)
            Staff[i].raiseSalary(100);
         for (i = 0; i < Staff.length; i++)
            Staff[i].print();

       }
```



```
class Employee implements Serializable
{  public Employee(String n, double s, Day d)
   {  name = n;
      salary = s;
      hireDay = d;
   }
   public Employee() {}
   public void print()
   {  System.out.println(name + " " + salary
         + " " + hireYear());
   }
   public void raiseSalary(double byPercent)
   {  salary *= 1 + byPercent / 100;
   }
   public int hireYear()
   {  return hireDay.getYear();
   }
```



```
private String name;
   private double salary;
   private Day hireDay;
}
class Manager extends Employee
{  public Manager(String n, double s, Day d)
   {  super(n, s, d);
      secretaryName = "";
   }
   public Manager() {}
   public void raiseSalary(double byPercent)
   {  // add 1/2% bonus for every year of service
      Day today = new Day();
      double bonus = 0.5 * (today.getYear() -
              hireYear());
      super.raiseSalary(byPercent + bonus);
   }
```



```
   public void setSecretaryName(String n)
   {  secretaryName = n;
   }

   public String getSecretaryName()
   {  return secretaryName;
   }

   private String secretaryName;
}
```

harry = new Employee("Harry Hacker",…..);
Manager carl = new Manager("Carl Cracker",….);
carl.setSecretary(harry);
Manager tony = new Manager("Tony Tester",….);
tony = setSecretary(harry);

Heap memory Layout for Staff, Employee and Manager objects.