Deen Dayal Upadhyaya College
Department of Computer Science
Workshop on Java J2SE

Dr. Rajni Bala
Associate Professor
Dept. of Computer Science
DDUC

Input/Output in JAVA

I/O streams

- Java performs I/O through streams.
- A stream is linked to a physical layer by java I/O system to make input and output operation in Java.
- A stream means continuous flow of data.

I/O Stream

- Java encapsulates Stream under **java.io** package.
- It defines two types of Streams

1. Byte Stream : It provides a convenient means for handling input and output of byte.
2. Character Stream : It provides a convenient means for handling input and output of characters. Character stream uses Unicode.

Byte Stream Classes

- Byte Stream is defined by using two abstract class at the top of hierarchy,

1. InputStream
2. OutputStream

These two abstract classes have several concrete classes that handle various devices such as disk files, arrays, network connection etc.

Some Byte Stream Classes

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Table 12-1. *The Byte Stream Classes*

Byte Streams

- These classes define several key method. Two most important methods are

1. read() : reads byte of data.
2. write() : writes a byte of data.

Character Stream

- Character stream is also defined by using two abstract class at the top of hierarchy.
- They are

1. Reader
2. Writer

Character Stream

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Table 12-2. *The Character Stream I/O Classes*

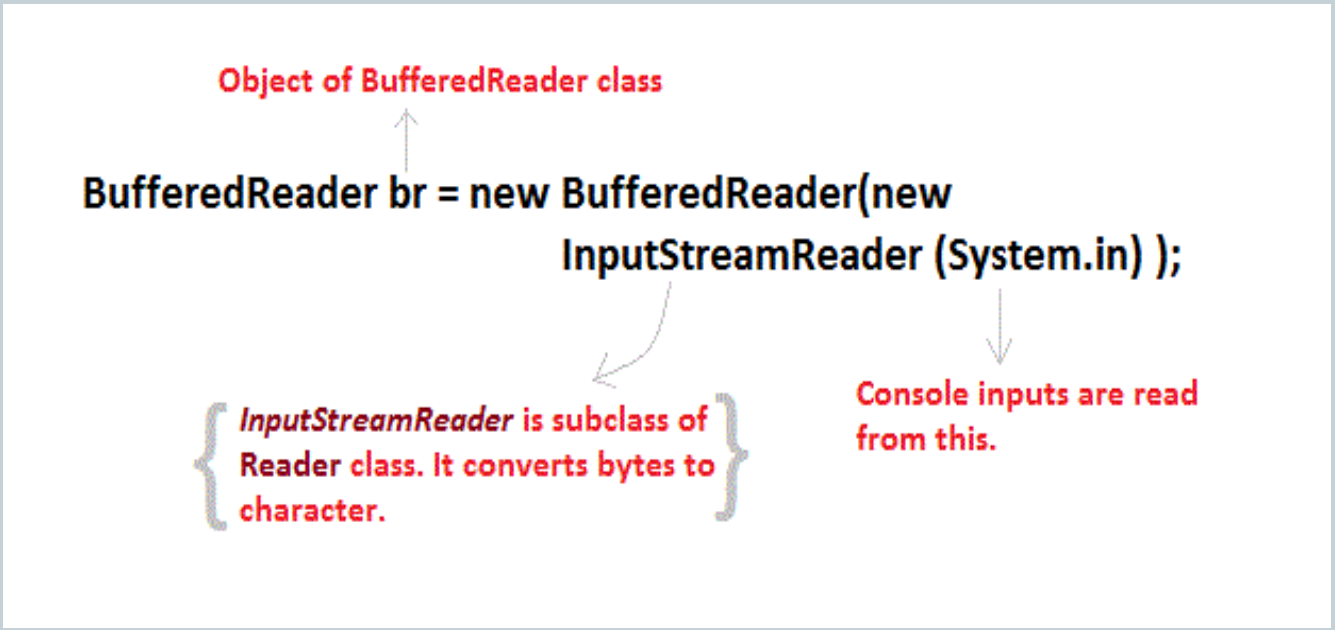
Character Stream

- Two of the most important methods are **read( )** and **write( )**, which read and write characters of data, respectively.
- These methods are overridden by derived stream classes.

Predefined Streams

- All java programs import java.lang
- This package defines a class called **System**
- **System** contains three predefined stream variables, **in**, **out**, and **err**. These fields are declared as **public** and **static** within **System**.
- **System.out** refers to the standard output stream. By default, this is the console. It is an object of InputStream
- **System.in** refers to standard input, which is the keyboard by default. It is an object of PrintStream
- **System.err** refers to the standard error stream, which also is the console by default. It is an object of PrintStream

Reading character



Reading Characters

```java
// Use a BufferedReader to read characters from the console.
import java.io.*;
class IoDemo
{
    public static void main(String args[]) throws IOException
    {
        char c;
        BufferedReader br = new  BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter characters, 'q' to quit.");
        // read characters
        do
        {
            c = (char) br.read();
            System.out.println(c);
        } while(c != 'q');
    }
}
```

Reading Strings

- readLine() - is a member of the **BufferedReader** class. Its general form :

String readLine( ) throws IOException

```
// Use a BufferedReader to read characters from the console.
import java.io.*;
//Read a string from console using a BufferedReader.
import java.io.*;
class IoDemo
{
    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"));
    }
}
```

Writing Console Output

- Console output is most easily accomplished with **print( )** and **println( )**.
- These methods are defined by the class **PrintStream** (which is the type of the object referenced by **System.out**)..
- **PrintStream** is an output stream derived from **OutputStream**, it also implements the low-level method **write( )**.
- The simplest form of **write( )** defined by **PrintStream** is:

  void write(int *byteval*)

- This method writes to the stream the byte specified by *byteval*. Although *byteval* is declared as an integer, only the low-order eight bits are written.

Using write function

```
// Demonstrate System.out.write().
class WriteDemo {
  public static void main(String args[]) {
    int b;

    b = 'A';
    System.out.write(b);
    System.out.write('\n');
  }
}
```

PrintWriter Class

- **PrintWriter** is one of the character-based classes. Using a character-based class for console output makes it easier to internationalize your program.

- **PrintWriter** defines several constructors. The one we will use is shown here:

PrintWriter(OutputStream *outputStream*, boolean *flushOnNewline*)

Here, *outputStream* is an object of type **OutputStream**, and *flushOnNewline* controls whether Java flushes the output stream every time a **println( )** method is called. If *flushOnNewline* is **true**, flushing automatically takes place. If **false**, flushing is not automatic.

- **PrintWriter** supports the **print( )** and **println( )** methods for all types including **Object**. Thus, you can use these methods in the same way as they have been used with **System.out**. If an argument is not a simple type, the **PrintWriter** methods call the object's **toString( )** method and then print the result.

Example

```
// 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);
  }
}
```

Reading from file

- **FileInputStream** and **FileOutputStream**, which create byte streams linked to files.

- To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor.

  FileInputStream(String *fileName*) throws FileNotFoundException
  FileOutputStream(String *fileName*) throws FileNotFoundException

  void close( ) throws IOException
  int read( ) throws IOException //returns -1 at the end
  void write(int *byteval*) throws IOException

Program to copy one file to another

```
import java.io.*;
class CopyFile
{
    public static void main(String args[]) throws IOException
    {
    int i;
    FileInputStream fin;
    FileOutputStream fout;
    try
    {
        // open input file
        try
        {
            fin = new FileInputStream(args[0]);
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Input File Not Found");
            return;
        }
        // open output file
        try
        {
            fout = new FileOutputStream(args[1]);
        }
        catch(FileNotFoundException e)
        {
            System.out.println("Error Opening Output File");
            return;
        }
```

```
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println("Usage: CopyFile From To");
        return;
    }
    // Copy File
    try
    {
        do
        {
            i = fin.read();
            if(i != -1) fout.write(i);
        } while(i != -1);
    }
    catch(IOException e)
    {
        System.out.println("File Error");
    }
    fin.close();
    fout.close();
    }
}
```

Methods by InputStream

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.
boolean markSupported( )	Returns **true** if **mark( )/reset( )** are supported by the invoking stream.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.

Methods by OutputStream

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int *b*)	Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with expressions without having to cast them back to **byte**.
void write(byte *buffer*[ ])	Writes a complete array of bytes to an output stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].

Table 17-2. *The Methods Defined by* OutputStream

File Class

- A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

- To create an object

  File(String *directoryPath*)
  File(String *directoryPath,* String *filename*)
  File(File *dirObj*, String *filename*)
  File(URI *uriObj*)

File functions

Method	Description
`f.exists()`	Returns true if file exists.
`f.isFile()`	Returns true if this is a normal file.
`f.isDirectory()`	true if "f" is a directory.
`f.getName()`	Returns name of the file or directory.
`f.isHidden()`	Returns true if file is hidden.
`f.lastModified()`	Returns time of last modification.
`f.length()`	Returns number of bytes in file.
`f.getPath()`	path name.
`f.delete()`	Deletes the file.
`f.renameTo(f2)`	Renames f to File f2. Returns true if successful.
`f.createNewFile()`	Creates a file and may throw IOException.

Program to demonstrate File

```
// 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");
  }
}
```

Directories

- A directory is a **File** that contains a list of other files and directories. When you create a **File** object and it is a directory, the **isDirectory( )** method will return **true**. In this case, you can call **list( )** on that object to extract the list of other files and directories inside.

```
// Using directories.
import java.io.File;
class FileDemo {
    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");
        }
    }
}
```

Using FilenameFilter

- To limit the number of files returned by the **list( )** method to include only those files that match a certain filename pattern, or *filter*. *One* must use a second form of **list( )**, shown here:

String[ ] list(FilenameFilter *FFObj*)

where *FFObj* is an object of a class that implements the **FilenameFilter** interface.

- **FilenameFilter** defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is:

boolean accept(File *directory*, String *filename*)

- The **accept( )** method returns **true** for files in the directory specified by *directory* that should be included in the list (that is, those that match the *filename* argument), and returns **false** for those files that should be excluded.

Example

- This program only files with have .html extension

```
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);
  }
}
```

```
// Directory of .HTML files.
import java.io.*;

class DirListOnly {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new OnlyExt("html");
    String s[] = f1.list(only);

    for (int i=0; i < s.length; i++) {
      System.out.println(s[i]);
    }
  }
}
```

Buffered Byte Streams

- For the byte-oriented streams, a *buffered stream* extends a stream class by attaching a memory buffer to the I/O streams.
- This buffer allows Java to do I/O operations on more than a byte at a time, hence increasing performance.
- Because the buffer is available, skipping, marking, and resetting of the stream becomes possible.
- The buffered byte stream classes are **BufferedInputStream, BufferedOutputStream**.
- **PushbackInputStream** also implements a buffered stream

Buffered Streams

- **BufferedInputStream** has two constructors:

  BufferedInputStream(InputStream *inputStream*)

  BufferedInputStream(InputStream *inputStream*, int *bufSize*)

- Other functions used

  read();

  skip();

  mark();

  reset();

Buffered Writer

- A **BufferedWriter** is a **Writer** that adds a **flush( )** method that can be used to ensure that data buffers are physically written to the actual output stream.

- **BufferedWriter** can increase performance by reducing the number of times data is actually physically written to the output stream.

- A **BufferedWriter** has these two constructors:

  BufferedWriter(Writer *outputStream*)

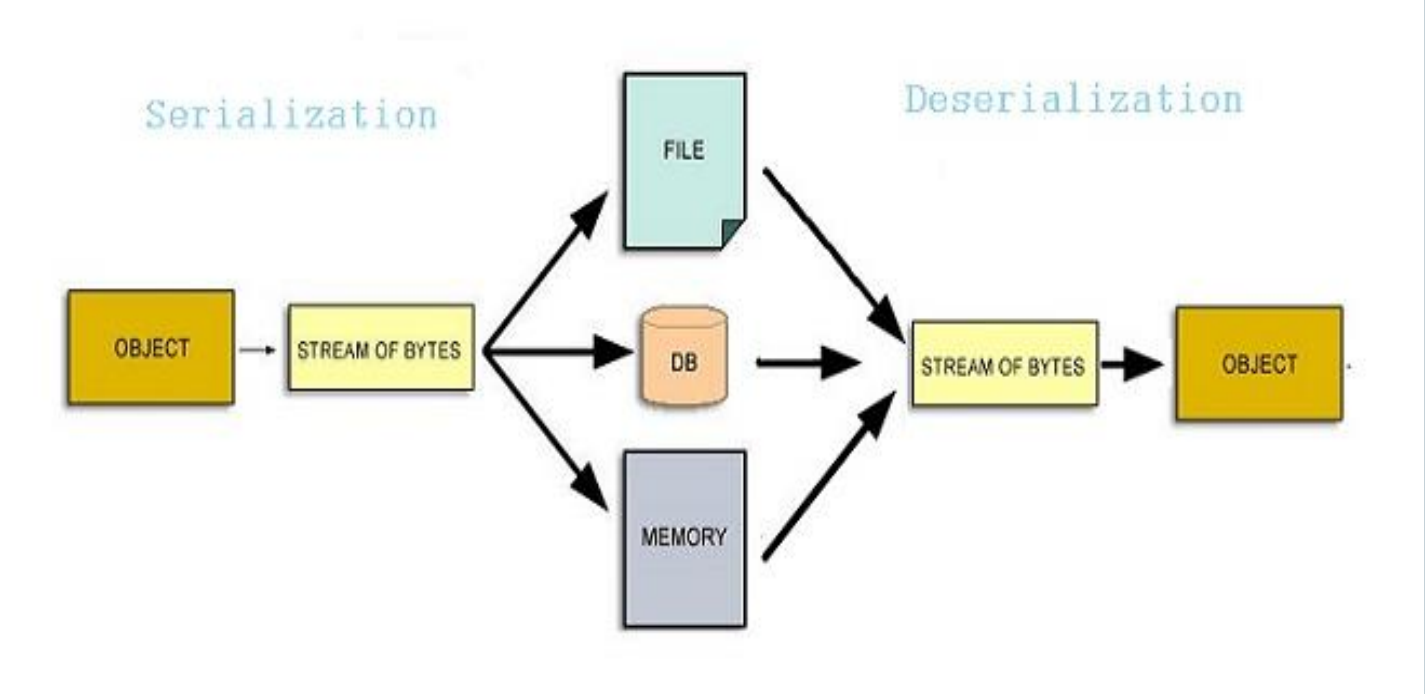
  BufferedWriter(Writer *outputStream*, int *bufSize*)

- The first form creates a buffered stream using a buffer with a default size. In the second, the size of the buffer is passed in *bufSize*.

Serialization/deserialization

- Serialization is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams.
- The reverse process of creating object from sequence of bytes is called deserialization.
- Serialization is also needed to implement Remote Method Invocation (RMI). RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it.

Serialization/deserialization



Serialization/deserialization

- Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph. There may also be circular references within this object graph. That is, object X may contain a reference to object Y, and object Y may contain a reference back to object X. Objects may also contain references to themselves.

- The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized.

- Similarly, during the process of deserialization, all of these objects and their references are correctly restored.

Serialization/deserialization

- A class must implement Serializable interface present in java.io package in order to serialize its object successfully.

- Java provides Serializable API encapsulated under java.io package for serializing and deserializing objects which include

  java.io.Serializable

  java.io.Externalizable

  ObjectInputStream

  ObjectOutputStream

Marker Interface

- Marker Interface is a special interface in Java without any field and method.
- It is used to inform complier that the class implementing it has some special behaviour or meaning.
- Some Marker interface are

  java.io.Serializable

  java.lang.Cloneable

  java.rmi.Remote

  java.util.RandomAccess

writeObject/ readObject

- writeObject () method of ObjectOutputSream class serializes an object and send it to the output stream.

  public final static writeObject(object x) throws IOException

- readObject() method of ObjectInputStream class references object out of stream and deserialize it.

  public final Object readObject() throws IOException, classNotFoundException

Serialization

```
import java.io.*;
class studentinfo implements Serializable
{
 String name;
 int rid;
 static String contact;
 studentinfo(string n, int r, string c)
 {
  this.name = n;
  this.rid = r;
  this.contact = c;
 }
}
```

```
class Test
{
 public static void main(String[] args)
 {
 try
 {
  Studentinfo si = new studentinfo("Abhi", 104, "110044");
  FileOutputStream fos = new FileOutputStream("student.ser");
  Objectoutputstream oos = new ObjectOutputStream(fos);
  oos.writeObject(si);
  oos.close();
  fos.close();
  }
  catch (Exception e)
  { e. printStackTrace(); }
 }
}
```

Deserialization

```
import java.io * ;
class DeserializationTest
{
 public static void main(String[] args)
 {
  studentinfo si=null ;
  try
  {
   FileInputStream fis = new FileInputStream("student.ser");
   ObjectOutputStream ois = new ObjectOutputStream(fis);
   si = (studentinfo)ois.readObject();
  }
  catch (Exception e)
   { e.printStackTrace(); }
  System.out.println(si.name);
  System.out. println(si.rid);
  System.out.println(si.contact);
 }
}
```

Transient keyword

- While serializing an object, if we don't want certain data member of object to be serialized we can mention it as transient . Transient keyword will prevent that data member from being serialized.
- Static members are never serialized as they are connected to class not object of class.

```
class studentinfo implements Serializable
{
 String name;
 transient int rid;
 static String contact;
}
```

ObjectOutputStream

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBoolean(boolean *b*)	Writes a **boolean** to the invoking stream.
void writeByte(int *b*)	Writes a **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBytes(String *str*)	Writes the bytes representing *str* to the invoking stream.
void writeChar(int *c*)	Writes a **char** to the invoking stream.
void writeChars(String *str*)	Writes the characters in *str* to the invoking stream.
void writeDouble(double *d*)	Writes a **double** to the invoking stream.
void writeFloat(float *f*)	Writes a **float** to the invoking stream.
void writeInt(int *i*)	Writes an **int** to the invoking stream.
void writeLong(long *l*)	Writes a **long** to the invoking stream.
final void writeObject(Object *obj*)	Writes *obj* to the invoking stream.
void writeShort(int i)	Writes a **short** to the invoking stream.

ObjectInputStream

int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
boolean readBoolean( )	Reads and returns a **boolean** from the invoking stream.
byte readByte( )	Reads and returns a **byte** from the invoking stream.
char readChar( )	Reads and returns a **char** from the invoking stream.
double readDouble( )	Reads and returns a **double** from the invoking stream.
float readFloat( )	Reads and returns a **float** from the invoking stream.
void readFully(byte *buffer*[ ])	Reads *buffer.length* bytes into *buffer*. Returns only when all bytes have been read.
void readFully(byte *buffer*[ ], int *offset*, int *numBytes*)	Reads *numBytes* bytes into *buffer* starting at *buffer*[*offset*]. Returns only when *numBytes* have been read.
int readInt( )	Reads and returns an **int** from the invoking stream.
long readLong( )	Reads and returns a **long** from the invoking stream.
final Object readObject( )	Reads and returns an object from the invoking stream.