Input and Output

The Java Platform supports different kinds of information sources and information sinks. A program may get data from an information source which may be a file on disk, a network connection, another program, standard input (e.g. keyboard), etc. Similarly a program may send data to an information sink which may be a file on disk, a network connection, another program, standard output (e.g. monitor), etc. In the Java I/O system, streams can be used to get and put information from an information source or to an information sink.

Streams support either sequential read from an information source or sequential write to an information source. Regardless of the type of information source or sink, read and write operations are almost same. In order to read data from an information source you first create a stream by specifying the information source. Then, you read data as long as there is more data. Finally, you close the stream. Similarly, in order to write data to an information sink you first create a stream by specifying the information sink. Then, you write data as long as there is more data. Finally, you close the stream.

There are actually two kinds of streams: byte oriented and character oriented. Byte oriented streams write bytes to a stream and read bytes from a stream. According to the direction of information flow, input streams or output streams are used for byte oriented streams. Input and Output streams are used when binary data is to be read and written. On the other hand, character oriented streams write characters to a stream and read characters from a stream. Readers and writers are used when using text based input and output (with proper encoding). According to the direction of information flow, readers or writers are used for character oriented streams.

Stream classes are implemented in java.io package. So, in order to use streams classes without package names, your program must have the following imports statement at the top:

```
import java.io.*;
```

Streams

`InputStream` and `OutputStream` are the superclasses for input and output streams, respectively. The specialized subclasses of these classes provide necessary implementation for different kinds of streams. Some of those specialized subclasses are as follows:

InputStream	OutputStream	Source/Sink Type
`FileInputStream`	`FileOutputStream`	A File on disk. File name can be passed to the constructor.
`ByteArrayInputStream`	`ByteArrayOutputStream`	A byte array in memory. A buffer of type byte array can be passed to the constructor.
`PipedInputStream`	`PipedOutputStream`	Another thread (connect PipedInputStream to PipedOutputStream before reading/writing bytes). Anything written from PipedOutputStream can be read from PipedInputStream.

`InputStream` defines three methods to read data `byte`s as:

Method	Description
`int read()`	reads next byte from the stream.
`int read(byte[] b)`	reads k<=b.length bytes from the stream and returns the total number of bytes read. If there is enough data, stream fills the buffer b, otherwise stream puts bytes read starting from b[0].
`int read(byte[] b, int offset, int length)`	reads k<=length bytes from the stream, puts those bytes starting from b[offset], and returns the total number of bytes read.

If the end of stream is reached, the read method returns `-1`. It is also possible to learn how many bytes can be read from the stream using `available()` method.
Similarly `OutputStream` defines three methods to write data `byte`s as:

Method	Description
`void write(int b)`	writes (byte)b to the stream.
`void write(byte[] b)`	writes b.length bytes in b to the stream.
`void write(byte[] b, int offset, int length)`	writes length bytes in b starting from b[offset] to the stream.

In the following, a simple example program to copy a file on disk to a new file using streams is shown:

```
import java.io.*;
public class CopyFile {
    public static void main(String[] args) throws
IOException{
        FileInputStream in = new
FileInputStream("input.txt");
```

```
        FileOutputStream out = new
FileOutputStream("output.txt");
        byte b[] = new byte[8192];
        int length;
        while((length = in.read(b))>0) out.write(b, 0,
length);
        in.close();
        out.close();
    }
}
```

Note that, stream related operations may raise an `IOException`. Therefore, you should either handle `IOExceptions` or specify `IOException` in method declaration.

Filtered Input and Filtered Output

The `InputStream` and `OutputStream` only provide necessary functionality for reading from and writing to streams. Filter streams, which accept an `InputStream` or an `OutputStream` as an argument to their constructors, enhance the functionality provided by these streams. Some of the filter streams that can be used are as follows:

Filter Stream	Constructor Arguments	Behavior
`DataInputStream`	`InputStream`	Allows reading primitive data types and Strings from input stream.
`BufferedInputStream`	`InputStream` and (optional) buffer size	Does buffered reading for improved reading performance.
`PushbackInputString`	`InputStream` and (optional) pushback buffer size	Enables unreading of read bytes
`DataOutputStream`	`OutputStream`	Allows writing primitive data types and Strings to input stream.
`BufferedInputStream`	`OutputStream` and (optional) buffer size	Does buffered writing for improved reading performance.
`PrintOutputString`	`OutputStream`, (optional) auto flashing enabled, (optional) encoding	Enables writing primitive data types and strings to output stream in text format.

An `InputStream` or `OutputStream` is usually wrapped by a filter stream to have a more useful interface. For example, the following code illustrates wrapping a `FileOutputStream` with a `DataOutputStream` to write primitive data types to a file:

```
DataOutputStream s = new DataOutputStream(new
FileOutputStream("test.dat"));
```

Then, it is possible to write primitive data types to the stream using `writeXXX(value)`, where `XXX` is the primitive data type name (like `Integer`, `Char`, `Float`, `Double`, etc).

Readers and Writers

`Reader`s and `Writer`s implement the similar functionality with `InputStream` and `OutputStream` with one exception: they work with characters instead of bytes. `Reader` and `Writer` are the superclasses for character oriented (text based) input and output streams, respectively. The specialized subclasses of these classes provide necessary implementation for different kinds of streams. Some of those specialized subclasses are as follows:

Reader	Writer	Source/Sink Type
`FileReader`	`FileWriter`	A File on disk. File name can be passed to the constructor.
`CharArrayReader`	`CharArrayWriter`	A char array in memory. A buffer of type char array can be passed to the constructor.
`StringReader`	`StringWriter`	String in memory. A string can be passed to the constructor of StringReader and an initial size can be passed to the constructor of StringWriter.
`PipedReader`	`PipedWriter`	Another thread (connect PipedReader to PipedWriter before reading/writing characters). Anything written from PipedWriter can be read from PipedReader.

Like `InputStream`, `Reader` defines three methods to read characters as:

Method	Description
`int read()`	reads next byte from the stream.
`int read(char[] b)`	reads k<=b.length characters from the stream and returns the total number of characters read. If there is enough data, stream fills the buffer b, otherwise stream puts characters read starting from b[0].
`int read(char[] b, int offset, int length)`	reads k<=length characters from the stream, puts those characters starting from b[offset], and returns the total number of characters read.

If the end of stream is reached, the read method returns `-1`. It is also possible to learn how many characters can be read from the stream using `available()` method.

`Writer` defines five methods to write characters and strings as:

Method	Description
`Void write(int b)`	writes (char)b to the stream.
`void write(char[] b)`	writes b.length characters in b to the stream.
`void write(char[] b, int offset, int length)`	writes length characters in b starting from b[offset] to the stream.
`void write(String s)`	writes string s to the stream.
`void write(String s, int offset, int length)`	writes length characters in string s starting from character at position offset to the stream.

It is also possible to have a more useful interface by wrapping readers and writers by other `Reader` and `Writer` classes. For example, by wrapping a reader with `BufferedReader` you can read strings in addition to characters and character arrays. For example, the following code wraps a `FileReader` in `BufferedReader` to read strings line by line and prints lines to the screen:

```
BufferedReader r = new BufferedReader(new
FileReader("textfile.txt"));
String s;
while((s=r.readLine())!=null) System.out.println(s);
```

`InputStreamReader` and `OutputStreamWriter` classes provide a mechanism to wrap `InputStream`s and `OutputStream`s in `Reader`s and `Writer`s for text based I/O.

Standard I/O

Java provides three streams for standard I/O: `System.in`, `System.out`, and `System.err`. Since `System.out` and `System.in` are prewrapped by `PrintStream`s, any primitive type and strings can be directly written to the console using `print(...)` and `println(...)` methods. However, `System.in` is in `InputStream` form, and in order to read strings from the standard input (e.g. keyboard) `System.in` is usually wrapped by `BufferedReader` and `InputStreamReader` (to convert a stream interface to reader interface) before reading. The following code illustrates the reading from standard input and writing to standard output using `System.in` and `System.out`:

```
import java.io.*;
public class Echo {
    public static void main(String[] args) throws Exception{
        BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
        String s;
        while(((s=in.readLine())!=null) && (s.length()>0))
            System.out.println(s);
    }
}
```

The above program echoes the lines entered from standard input

to standard output until an empty line is entered.

Object Serialization

Object serialization enables you to convert an object into series of bytes, and reconstruct the object from series of bytes. The instances of classes implementing `Serializable` interface can automatically be converted to bytes and can automatically reconstructed from the bytes. To make the a class serializable, you could just add implements `Serializable` to the class declaration as:

```
class MySerializableClass implements Serializable {
    ...
}
```

Since there is no method in `Serializable` interface, you do not need to implement anything. However, care should be taken when declaring member variables of the class. It may not be possible to serialize some sensitive data (such as a file handle). For such variables you can simply add `transient` keyword to avoid serialization of that variable as:

```
transient DataType VariableName;
```

Objects implementing the `Serializable` interface can be written to and read from streams using `ObjectOutputStream` and `ObjectInputStream` wrappers as:

```
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("output.dat"));
ObjectInputStream in = new ObjectInputStream("input.dat"));
Object obj;
while((obj=in.readObject())!=null) out.writeObject(obj);
in.close();
out.close();
```