I/O in Java - a very quick overview

I/O operations are a little more basic in Java than C++. Basically, there are a hierarchy of streams (sequence of data) and the type of stream used is largely dependent upon the data requirements of the application. We shall concentrate mainly on two types of stream

- Character based streams - reading and writing text.
- Object based streams - reading and writing binary data.

The two main classes of stream are input stream and output stream. Both these main types are further divided into character streams and byte streams with further division into buffered and un-buffered streams. Furthermore, file I/O is performed using *FileInput* and *FileOutput* streams.

Output Streams

Output streams are written using objects called writers. The general idea is to create a stream then attach a **PrintWriter** to it. Once this is done we use the **PrintWriter** to send data to the stream. You can view the stream **System.out** as a stream with a **PrintWriter** already attached to it.

Text Output

If you wish to write text to a file you should

1. Create a **FileWriter**
2. Attach a **PrintWriter** to this.

Both classes are in the **java.io** package that should be imported in the program if these two classes are used. The **FileWriter** object is created to let us output things to a file whereas the **PrintWriter** object is created so that we can print to the associated file using **println**. The way one does this is illustrated in ex3 below. The relevant lines of code are:

```
FileWriter fout = new FileWriter(fileName,false);
PrintWriter pout = new PrintWriter(fout,true);
```

The first line creates a stream called “fout” connected to the file whose name given in the String parameter “filename”. The second parameter determines if the data is appended to the file (parameter is true) or if the data replaces an existing file (parameter false). If a file with the specified name doesn't already exist, then a new file is created. The second line connects a new **PrintWriter** called pout to the stream “fout”. Once created this **PrintWriter** is used to access the file (see code in ex3). The second parameter determines if auto flushing is used.

Notice the keyword **new** used to create objects. In Java **new** is always used to create objects and you will find yourself using it much more than in C++.

A problem with file access is that errors may occur during the creation of the file or subsequently when reading/writing from/to the file. Java I/O procedures notify the program that an error occurs by throwing **(**i.e, generating) an **IOException** object. This exception must be directed to an error handling mechanism (that is the exception must be caught and dealt with). Therefore, we need to be able to capture an IO error if one is generated. The **try {} catch(...) {}** block provides us with a way to capture an error and process what to do if an error occurs. In ex3 if an IO Exception occurs it will display an error message which says why things went wrong. Exceptions provide us with a number of methods. The **getMessage()** method returns a string describing what the reason of the error is. We can also get the exception to display information about what class and method caused the problem using the **printStackTrace().** Implement and run ex3.java

```
import java.io.*;

class ex3 {

        public static void main(String[] args)  {
            if (args.length != 3) {
                    System.err.println("Usage: java ex3 <file> <from> <to>");
                    System.err.println("you must supply three argument values");
                    System.exit(0);
            } else {
                     String fileName  = args[0];                 // the first argument
                     int from = Integer.parseInt(args[1]);       // the second argument
                     int to = Integer.parseInt(args[2]);         // third argument

                     // now create the filestream and connect PrintWriter
                    try {
                          FileWriter fout = new FileWriter(fileName,false);
                          PrintWriter pout = new PrintWriter(fout,true);

                          //write to the file
                          for (int i = from; i < to; i++ ) {
                             pout.println(i);
                          }
                          pout.close();        // close the stream

                     } catch (IOException e) {
                          System.err.println("Error! - " + e.getMessage());
                     }
            }
        }
}
```

Binary Output

If you wish to write an object to a file in binary form, then you should:

- create a **FileOutputStream** as before
- create an **ObjectOutputStream** from the **FileOutputStream**
- use the **ObjectOutputStream** to write to the file.

The listing shown below is a new class called ex3A which behaves as class ex3 above but writes to the file in binary. Changes to the code are highlighted in a different colour. Implement and run ex3A.java

```
import java.io.*;

class ex3A {
        public static void main(String[] args)  {
            if (args.length != 3) {
                    System.err.println("Usage: java ex3 <file> <from> <to>");
                    System.err.println("you must supply three argument values");
                    System.exit(0);
            } else {
                     String fileName  = args[0];                 // the first argument
                     int from = Integer.parseInt(args[1]);       // the second argument
                     int to = Integer.parseInt(args[2]);         // third argument
```

```
                // now create the filestream and connect PrintWriter
                try {
                    FileOutputStream    fout = new FileOutputStream(fileName,true);
                    ObjectOutputStream sout = new  ObjectOutputStream(fout);

                     //write to the file
                     for (int i = from; i < to; i++ ) {
                                sout.writeInt(i);
                                sout.flush();
                     }
                     sout.close();        // close the stream

                } catch (IOException e) {
                     System.err.println("Error! - " + e.getMessage());
                }
            }
        }
}
```

ObjectOutputStreams do not have a **print** or **println** function. Instead they have a collection of write-like functions one for each of the basic types. For example, **writeInt** for writing integers, **writeFloat** for writing floats etc. Use the **writeObject** to write all the non-basic types including String to the stream.

Example code fragment to write an **int** followed by a **String** followed by a **double** to the stream

```
sout.writeInt(3);
sout.writeObject("This is a string");
sout.writeDouble(4.9876);
sout.flush();
```

Notice that **ObjectOutputStreams** are not auto-flushing so that the flush function must be called after each write to flush the stream. To find out exactly what **flush()** does, look up the method in the Java Documentation

Again, possible Exceptions must be caught (this is true for all file I/O operations) using a try catch block.

Exception handling - the try .. catch block

the following code structure represents Java's exception handling mechanism.

```
try
{...
      // code that might generate an Exception.
}
catch (IOException e)
{
      // if the Exception generated is the same type as IOException
      // then run the following code to handle the Exception stored in variable e
}
```

If an IO error occurs when executing the code in the **try** block an **IOException** is generated. The Java system will cease executing the code in the try block and will look for a matching exception handler (i.e., catch block). If a suitable catch block exists (the catch Exception type matches the exception generated) the code in the catch block will be executed. All that happens in the example above is that the Exception is converted to a String and printed out. The program will then terminate.

If no matching handler (catch block) is found the exception is propagated up through the program until either a matching catch block is found or the top level of the program is reached when this occurs the program terminates.

If no error occurs and all the code in the try block is successfully executed any catch block(s) are ignored and execution continues at the statement following the **try..catch** block.

The type IOException represents a variety of different exceptions which can be caught individually. (See Java Documentation) if a finer error handling control is required.

Input Streams

Normally data is not read from the input stream directly. A reader is generally connected to the stream and then used to access the input stream.

Text Input Streams

If the data is from a text file, then you should:

- create a **FileReader** connected to the file
- connect **FileReader** to a **BufferedReader** for increased functionality.

Reading from text files is very low level even with a **BufferedReader**. Normally you can read either:

- a single character using the **read()** function
- Read characters into a portion of an array using **read(char[] cbuf, int off, int len)**
- Read line of text into a String using **readLine().**

Typically a program will use **readLine()** to obtain the next line of text as a String and then either parse the resulting line into its constituent parts or use the functionality of the String class to search for substrings etc. The complexity of this parsing process will depend upon the complexity of the data layout in the file. Because everything is read in as a String Java provides several useful conversion functions to "parse" Strings into other basic types.

- Byte.parseByte(String) returns the byte represented by the String
- Short.parseShort(String) returns the short represented by the String
- Integer.parseInt(String) returns the int represented by the String
- Long.parseLong(String) returns the long represented by the String

Alternatively, you can use the following:

- Boolean.valueOf(String) returns the boolean represented by the String
- Float.valueOf(String) returns the float represented by the String
- Double.valueOf(String) returns the double represented by the String

If a String representing a single character is read in; then the character may be obtained by extracting it from the 0^{th} position in the String using the **charAt(int)** function. Its value as a character will be **charAt(0)**.

If the file structure is such that each line of the file contains a single data item of the same type, then the code to read the file is quite straightforward and comprises of a while loop. Because **readLine()** returns the value **null** (In Java this represents no value) the code to read the file can be included in the condition test of the while loop which gives a very neat implementation as highlighted in the fragment of code shown below.

```
FileReader      fin = new FileReader(fileName);
BufferedReader  din = new BufferedReader(fin);

String line = null;       //declare variable to store a line of text

while ((line = din.readLine()) != null) {
      // here we have read in a line of text
      // now parse line to extract data and print it out to the screen
}
din.close()    // close file when we have finished
```

This structure is used in the ex4 below that reads in numbers from a file and calculates their total. Implement and run ex4.java.

```
import java.io.*;

class ex4 {
        public static void main(String[] args)  {
                if (args.length != 1) {
                   System.err.println("Usage: java ex4 <file>");
                   System.err.println("you must supply the name of the file");
                   System.exit(0);
                } else {
                   String  fileName = args[0];          // the first argument

                   // now create the filestream and connect PrintWriter
                   // the value true enables autoflushing
                   try {
                           FileReader         fin = new FileReader(fileName);
                           BufferedReader din = new BufferedReader(fin);

                           System.out.println("Data in file " + fileName + " is:  ");

                           //read from the file
                           String line = null;      // line of text
                           int sum = 0;               // running total of  numbers

                           while ((line = din.readLine()) != null)  {
                             // here we have read in a line of text
                             // now parse line to extract data and print it out to the screen

                             int value = Integer.parseInt(line);
                             sum += value;

                             System.out.println("\t" + value);
                           }
                           System.out.println("Sum of numbers in file was " + sum);
                           din.close();         // close the stream
                    } catch (IOException e) {
                           System.err.println("Error! - " + e.getMessage());
                    }
                }
        }
  }
```