Stream Classes and File I/O

A *stream* is any input source or output destination for data. The Java API includes about fifty classes for managing input/output streams. Objects of these classes can be instantiated and methods exist to perform "actions" on these objects. The most basic actions are sending data to a stream or receiving data from a stream. In this section, we'll examine the basic operation of streams, focusing on the standard I/O streams and on file streams. For a detailed look at Java's stream classes, see Chapter 12 in Arnold and Gosling's *The Java Programming Language*.

A Java program is reasonably well served by its default state when execution begins. Three streams are setup and ready to go. There are two output streams, identified by the objects `System.out` and `System.err`, and one input stream, identified by the object `System.in`. These objects are defined as public data fields in the `System` class of the `java.lang` package (see Table 1).

Table 1. Data Fields in the `System` Class

Variable	Type	Purpose
`err`	`PrintStream`	the "standard error" output stream. This stream is already open and ready to accept output data.
`in`	`InputStream`	the "standard input" stream. This stream is already open and ready to supply input data.
`out`	`PrintStream`	the "standard output" stream. This stream is already open and ready to accept output data.

As noted in the table, the streams are "open and ready" when a Java program begins execution. To use these objects in a program, they must be prefixed with "`System.`" to identify where they are defined. This is similar to our use of the prefix "`Math.`" when using methods of the `Math` class.

The `err` and `out` objects are instances of the `PrintStream` class and the `in` object is an instance of the `InputStream` class. As you might guess, methods in these classes provide the means to input or output data. As well, the default devices for these streams are the keyboard for input and the host system's CRT display for output.

Two output streams are provided to separate "normal" output from error messages. Normal output can be redirected to a file, as demonstrated earlier, and this is a useful service. However, if errors occur during program execution, error messages appear on the host system's CRT display, even if output was redirected to a file. In most cases, it is useful to separate these two forms of output, and, so, "standard output" and "standard error" streams exist.

Let's begin by examining the standard output stream.

The Standard Output Stream

We have already used the standard output stream extensively. Variations of the following statement

```
System.out.println("Hello");
```

appear frequently in our demo programs. The method `println()` is called via an instance of the `PrintStream` class, in this case the object variable `out`. The methods of the `PrintStream` class are summarized in Table 2.

Table 2. Methods of the `PrintStream` Class

Method	Purpose
Constructors	
`PrintStream(OutputStream out)`	creates a new print stream. This stream will not flush automatically; returns a reference to the new `PrintStream` object
`PrintStream(OutputStream out, boolean autoFlush)`	creates a new print stream. If `autoFlush` is true, the output buffer is flushed whenever (a) a byte array is written, (b) one of the `println()` methods is invoked, or (c) a newline character or byte ('\n') is written; returns a reference to the new `PrintStream` object
Instance Methods	
`flush()`	flushes the stream; returns a `void`
`print(char c)`	prints a character; returns a `void`
`println()`	terminates the current line by writing the line separator string; returns a `void`
`println(char c)`	prints a character and then terminates the line; returns a `void`

Note that the `print()` and `println()` methods are overloaded. They can accept any of the primitive data types as arguments, as well as character arrays or strings.

If `println()` is called without an argument it serves only to terminate the current line by outputting a newline character ('`\n`'). The method `print()` outputs its argument as a series of characters without automatically sending a newline character. On some systems, the `print()`must be followed by the `flush()` to force characters to print immediately. Otherwise, they are held in an output buffer until a subsequent newline character is printed. This behaviour is established when the output stream is opened, as seen in the descriptions of the constructor methods. The example programs in this text were written using *Java 2* on a host system running the *Windows98* operating system. The `autoFlush` option is "on", and, so the `flush()` method is not needed. It may be necessary to follow `print()` with `flush()` on other systems.

As demonstrated in earlier, the standard output stream may be redirected to a disk file by appending "`>`" followed by a filename when the program is invoked.

The Standard Input Stream

Unfortunately, the behaviour of Java programs is not quite as clean for the standard input stream as it is for the standard output stream. The object variable "`in`" is defined in the `System` class as an instance of the `InputStream` class. As is, the standard input stream is setup to read raw, unbuffered bytes from the keyboard. Such low-level control may be necessary in some cases, but the result is inefficient and provides poor formatting of input data. For example, it forces us to deal explicitly with keystrokes such as "backspace" and "delete". For most purposes, this is a nuisance. Earlier, we illustrated a convenient way to re-package the standard input stream:

```
BufferedReader stdin =
   new BufferedReader(new InputStreamReader(System.in), 1);
```

The object `System.in` appears above as an argument to the constructor method for the `InputStreamReader` class. The resulting object — an instance of the `InputStreamReader` class — in turn appears as an argument to the constructor method for the `BufferedReader` class. A reference to the instantiated object is assigned to the object variable `stdin`.

There is a long story and a short story to the above syntax. If you want all the details on the `InputStream` class and the `InputStreamReader` class, see the Java API documentation. The short story is that our access to the standard input stream now occurs via the object variable `stdin` which is an instance of the `BufferedReader` class in the `java.io` package. So, the methods of the `BufferedReader` class are of primary concern to us. These are summarized in Table 3.

Table 3. Methods of the `BufferedReader` Class

Methods	Description
Constructor Methods	
`BufferedReader(Reader in)`	creates a buffered character-input stream that uses a default-sized input buffer; returns a reference to the new object
`BufferedReader(Reader in, int sz)`	creates a buffered character-input stream that uses an input buffer of the specified size; returns a reference to the new object
Instance Methods	
`close()`	returns a `void`; closes the stream
`readLine()`	returns a `String` object equal to a line of text read

For our purposes, we only need the `readLine()` method. Each line typed on the keyboard is returned as a `String` object. For example,

```
String s = stdin.readLine();
```

If the line contains a representation of a primitive data type, it is subsequently converted using a "parsing" method of a wrapper class. If an integer value is input from the keyboard, the inputed string is converted to an `int` as follows:

```
int i = Integer.parseInt(s);
```

Since the `String` variable `s` isn't needed, the two statements above can be combined:

```
int i = Integer.parseInt(stdin.readLine());
```

If the line contains multiple words, or tokens, they are easily separated using the `StringTokenizer` class. This was demonstrated earlier.

We will use the `close()` method later when we learn to access disk files for input using the `BufferedClass`. The standard input stream is automatically closed upon program termination.

File Streams

The streams used most often are the standard input (the keyboard) and the standard output (the CRT display). Alternatively, standard input can arrive from a disk file using "input redirection", and standard output can be written to a disk file using "output redirection". (Redirection was discussed earlier.) Although I/O redirection is convenient, there are limitations. For one, redirection is "all or nothing". It is not possible, for example, to read data from a file using input redirection *and* receive user input from the keyboard. As well, it is not possible to read or write multiple files using input redirection. A more flexible mechanism to read or write disk files is available through Java's *file streams*.

The two file streams presented here are the *file reader stream* and the *file writer stream*. As with the standard I/O streams, we access these through objects of the associated class, namely the `FileReader` class and the `FileWriter` class. Unlike the standard I/O streams, however, we must explicitly "open" a file stream before using it. After using a file stream, it is good practice to "close" the stream, although, strictly speaking, this is not necessary as all streams are automatically closed when a program terminates.

Reading Files

Let's begin with the `FileReader` class. As with keyboard input, it is most efficient to work through the `BufferedReader` class. If we wish to read text from a file named `foo.txt`, it is opened as a file input stream as follows:

```
BufferedReader inputFile =
   new BufferedReader(new FileReader("foo.txt"));
```

The line above "opens" `foo.txt` as a `FileReader` object and passes it to the constructor of the `BufferedReader` class. The result is a `BufferedReader` object named `inputFile`. We have used the `BufferedReader` class extensively already for keyboard input. Not surprisingly, the same methods are available for `inputFile` as for `stdin` in our earlier programs. To read a line of text from `junk.txt`, we use the `readLine()` method of the `BufferedReaderClass`:

```
String s = inputFile.readLine();
```

Note that `foo.txt` is *not* being read using input redirection. We have explicitly opened a file input stream. This means the keyboard is still available for input. So, for example, we could prompt the user for the name of a file, instead of "hard coding" the name in the program:

```
// get the name of a file to read
System.out.print("What file do you want to read: ");
String fileName = stdin.readLine();

// open the file for reading
BufferedReader inputFile =
   new BufferedReader(new FileReader(filename));

// read the file
String s = inputFile.readLine();
```

Note, above, that keyboard input *and* file input take place in the same program. Once finished with the file, the file stream is closed as follows:

```
inputFile.close();
```

Some additional file I/O services are available through Java's `File` class, which supports simple operations with filenames and paths. For example, if `fileName` is a string containing the name of a file, the following code checks if the file exists and, if so, proceeds only if the user enters "y" to continue.

```
File f = new File(fileName);
if (f.exists())
{
   System.out.print("File already exists. Overwrite (y/n)? ");
   if(!stdin.readLine().toLowerCase().equals("y"))
      return;
}
```

Let's move on to an example. Figure 1 shows a simple Java program to open a text file called `temp.txt` and to count the number of lines and characters in the file.

```java
import java.io.*;

public class FileStats
{
   public static void main(String[] args) throws IOException
   {
      // the file must be called 'temp.txt'
      String s = "temp.txt";

      // see if file exists
      File f = new File(s);
      if (!f.exists())
      {
         System.out.println("\'" + s + "\' does not exit. Bye!");
         return;
      }

      // open disk file for input
      BufferedReader inputFile =
         new BufferedReader(new FileReader(s));

      // read lines from the disk file, compute stats
      String line;
      int nLines = 0;
      int nCharacters = 0;
      while ((line = inputFile.readLine()) != null)
      {
         nLines++;
         nCharacters += line.length();
      }

      // output file statistics
      System.out.println("File statistics for \'” + s + “\'...");
      System.out.println("Number of lines = " + nLines);
      System.out.println("Number of characters = " + nCharacters);

      // close disk file
      inputFile.close();
   }
}
```

Figure 1. `FileStats.java`

A sample dialogue with the program follows (user input is underlined):

```
PROMPT>java FileStats.java
File statistics for 'temp.txt'...
Number of lines = 48
Number of characters = 1270
```

Here's a programming challenge for you: Make a copy of `FileStats.java` and name the new file `FileStats2.java`. Modify the program so that the user is prompted to enter the name of the file to open. This makes more sense than to hard code the file's name within the program.

Writing Files

To open a file output stream to which text can be written, we use the `FileWriter` class. As always, it is best to buffer the output. The following sets up a buffered file writer stream named `outFile` to write text into a file named `save.txt`:

```
PrintWriter outFile =
   new PrintWriter(new BufferedWriter(new FileWriter("save.txt"));
```

Once again, we have packaged the stream to behave like one Java's standard streams. The object `outFile`, above, is of the `PrintWriter` class, just like `System.out`. If a string, `s`, contains some text, it is written to the file as follows:

```
outFile.println(s);
```

When finished, the file is closed as expected:

```
outFile.close();
```

Reading and write disk files, as just described, is plain and simple. We use the same methods as for reading the keyboard (the standard input) or for writing to the CRT display (the standard output). Of course, situations may arise with disk files that are distinctly different, and these should be anticipated and dealt with as appropriate. Earlier, we used the `exists()` method of the `File` class to ckeck if a file exists before attempting to open it, and the same approach may be used here. However, if we open and write data to an existing file, what happens to the previous contents? The old file is truncated and the previous contents are overwritten. In many situations, this is undesirable, and a variety of alternative approaches may be taken. The `FileWriter` constructor can be used with two arguments, where the second is a `boolean` argument specifying an "append" option. For example, the expression

```
new FileWriter("save.txt", true)
```

opens `save.txt` as a file output stream. If the file currently exists, subsequent output is appended to the file.

Another possibility is opening an existing *read-only* file for writing! In this case, the program terminates with an "access is denied" exception occurs. This can be caught and dealt with in the program, and we'll learn how to do so later.

Figure 2 shows a simple Java program to read text from the keyboard and write the text to a file called `junk.txt`.

```java
import java.io.*;

class FileWrite
{
   public static void main(String[] args) throws IOException
   {

      // open keyboard for input (call it 'stdin')
      BufferedReader stdin =
         new BufferedReader(new InputStreamReader(System.in));

      // Let's call the output file 'junk.txt'
      String s = "junk.txt";

      // check if output file exists
      File f = new File(s);
      if (f.exists())
      {
         System.out.print("Overwrite " + s + " (y/n)? ");
         if(!stdin.readLine().toLowerCase().equals("y"))
            return;
      }

      // open file for output
      PrintWriter outFile =
         new PrintWriter(new BufferedWriter(new FileWriter(s)));

      // inform the user what to do
      System.out.println("Enter some text on the keyboard...");
      System.out.println("(^z to terminate)");

      // read from keyboard, write to file output stream
      String s2;
      while ((s2 = stdin.readLine()) != null)
         outFile.println(s2);

      // close disk file
      outFile.close();
   }
}
```

Figure 2. `FileWrite.java`

A sample dialogue with this file follows (user input is underlined):

```
PROMPT>java FileWrite
Enter some text on the keyboard...
(^z to terminate)
Happy new year!
^z
PROMPT>
```

Note that `^z` means hold the Control key down while pressing `z`. (On unix systems, `^d` must be entered.)

If the program is immiately run again, then the following dialogue results:

```
PROMPT>java FileWrite
Overwrite junk.txt (y/n)? y
Enter some text on the keyboard...
(^z to terminate)
Happy birthday!
^z
PROMPT>
```

Since the file `junk.txt` was created on the first pass, it already exists on the second. Our code (see lines 15-22) provides a reasonable safety check before proceeding. Since the user entered "y", the program proceeds; however, the previous contents are overwritten.

There are several ways to demonstrate the "exceptional conditions" that may arise when working with disk files, and these are worth considering. Once the file `junk.txt` exists, make it "read only" and re-run `FileWrite` The program will terminate with the "access is denied" message noted earlier.[1] Another interesting demonstration is to open `junk.txt` in an editor, and then exectute `FileWrite`. The behaviour may vary from one editor to the next, but we'll leave this for you to explore.

Here's a programming challenge for you: Make a copy of `FileWrite.java` and name the new file `FileWrite2.java`. Modify the program such that `junk.txt` is opened in append mode. Compile the new program, then run it a few times to convince yourself that the file contents are preserved from one invocation to the next.

Class Hierarchy - Stream Classes

Earlier, we presented a class hierarchy showing Java's eight wrapper classes, the `Math` class, the `String` class, and the `StringTokenizer` class. In our present discussion of stream classes, several classes have been mentioned, and, of course, each holds a position in Java's overall class hierarchy. This is illustrated Figure 3. A few additional classes are shown that were not discussed. (You may want to explore these on your own.)

[1] To make `junk.txt` "read only", do the following. On *Windows* systems, locate the file in Windows Explorer then right-click on it and tick the "read-only" box. On unix systems, enter `chmod 444 junk.txt`.

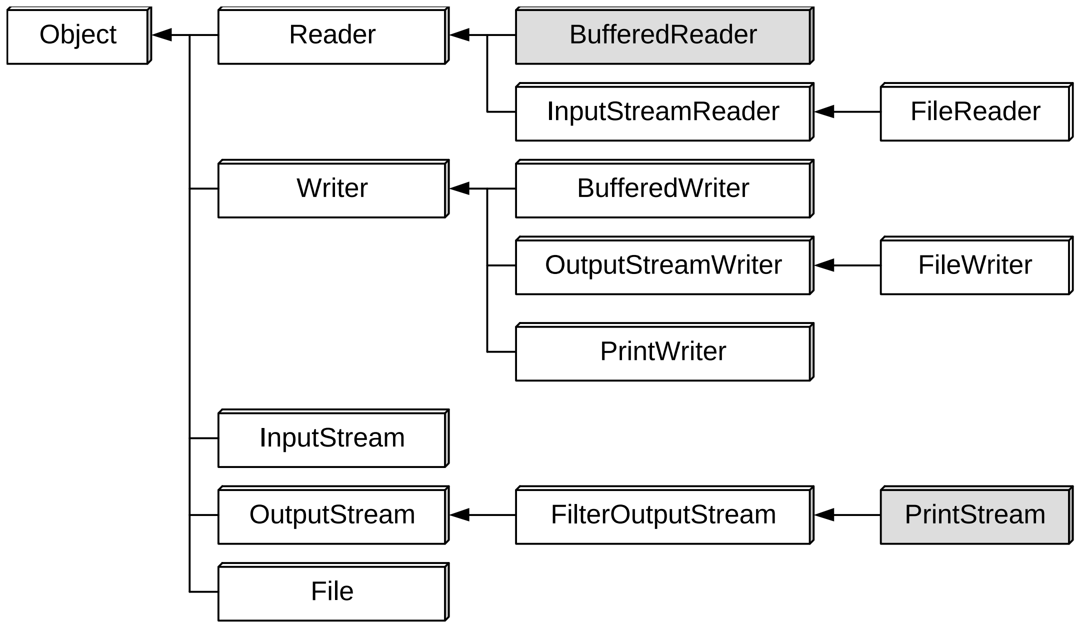

Figure 3. Class hierarchy for stream classes

The objects used in our demo programs for keyboard input and for output to the host systems' CRT are `stdin` and `System.out`. `stdin` is an object of the `BufferedReader` class and `System.out` is an object of the `PrintStream` class. These classes are highlighted in Figure 3. As we are well aware, instance methods of a class are accessed via instances variables of that class. So, for example the instance method `println()` of the `PrintStream` class can be accessed, or called, through `System.out`. As well, instance methods of a superclass may be accessed by an object of a subclass. (Recall that a subclass inherits characteristics, such as methods, from its superclass). So, for example, instance methods of the `PrintStream`, `FilterOutputStream`, `OutputStream`, and `Object` classes are all accessible via `System.out`, because `PrintStream` inherits from `FilterOutputStream`, which inherits from `OutputStream`, which, in turn, inherits from `Object`.

Now, if you are inclined to search through the Java API documentation to see what methods are available and to write small programs to make methods "further up" act on objects "further down", congratulations: You have caught the "Java bug" — not in the programming-error sense, but in the Java-is-fun sense. And, by all means, please do. The main point here, however, is to alert you to — and keep you thinking about — the overall structure and organization of the Java API.