13 File Output and Input

This section contains additional material regarding writing and reading from files. It is not needed for the second year course, but is essential for future programming. It is strongly suggested that you read and work through this section in your own time to develop your own programming skills and understanding of the language.

13.1 Introduction

To make programs really useful we have to be able to input and output data in large machine-readable amounts, in particular we have to be able to read and write to files. Since JAVA has extensive network and WEB support built in the input and output system is very complex, but for simple applications this can be simplified to a few recipes.

13.2 The Basics

There are two types of input and output schemes in JAVA these being (a) character based and (b) binary data based[1]. In this section we will only consider the character based files where output files typically hold tables of numbers to be printed, and input files are created by the editor.

The JAVA file model contains three parts, these being

1. The file on disc which is characterised by its name and other attributes such as read/write permission, creation date etc.
2. The high-level Reader/Writer classes that implement simple to use methods to read/write Strings and Lines.
3. The Stream classes that connect the high level Reader/Writer classes to the actual files.

In this section we will give a recipe for connecting these three together, so that you will only have to deal with the high-level Reader/Writer classes.

13.3 Output with the `PrintWriter` class

The `PrintWriter` class implements simple character based file output as shown in the example below.

```
import java.io.*;                                              // (1)
public class PrinterTest {
  public static void main(String args[]) throws IOException { // (2)
   String fileName = "mydata.data";                            // (3)
   PrintWriter output = new PrintWriter(
       new FileWriter(fileName));                              // (4)

   output.println("John Xavier Smith");                        // (5)
```

[1]There also the distinction between sequential and random access which will not be discussed here.

```
        output.println("Holland House Annex");                    // (6)
        output.println("Pollock Hall of Residence");              // (7)
        output.println("Edinburgh");                              // (8)

        output.close();                                           // (9)
        System.exit(0);
      }
  }
```

Looking at this Line-by-line,

(1) Import the `java.io` classes, this make the file manipulation *classes* available in your programme.

(2) Start of main program. You must add `throws IOException` to warn the compiler that this program may generate a exception (or error), from the `IO` classes. (See the advanced sections of book on how to `catch` exceptions.)

(3) Create a `String` holding the name of the file we wish to write to.

(4) Create a `FileWriter` object to write to the specified file and also create a `PrintWriter` which implements easier to call methods. (Use as recipe).

Note if the specified file does not exist, it is created, if it does exist it is *overwritten* and the contents of the original file lost.

(5-8) Write a series of lines to the output file. Note the syntax of `println` is identical to that of the `Display` object used throughout the course.

(9) Close the file, this also causes the contents to be flushed to disc. Failure to call this will result in some, if not all, of the contents of the files being lost.

So once the `PrintWriter` is created and attached to the output file the actual writing of the data is very simple, you basically send `String`s that are appended into the current position in the file. There *are* other output methods associated with `PrintWriter` to output `String`s, `int`s, `double`s, `boolean`s, with and without newlines, however for most programs the above strategy of,

1. Form an output line as a `String`.
2. output with `println` *or* `printf`.

is what you actually need.

Aside: In your early programs you used `System.out.println()` *to output* `String`*s to the terminal screen. Here you were actually using a pre-opened* `PrintWriter` *connected to the "standard output stream" (*`System.out`*) which for interactive programs goes to the terminal window that started them.*

For a fuller example, see `CosPrinter.java` on the examples page which outputs to file the x, y coordinate pairs of a cos() graph with input parameters via the `Display` class.

13.4 Input with the `BufferedReader` class

Reading in of data from a file is slightly more complex, the simplest interface being offered by `BufferedReader`[2] which is again best explained by an example that reads a line at a time from the file and outputs it to the screen.

```
import java.io.*;
public class ReaderTest {
   public static void main(String args[]) throws IOException {
   String fileName = "mydata.data";
   BufferedReader input = new BufferedReader(                    // (1)
        new FileReader(fileName));

   String line;                                                  // (2)
   while (( line = input.readLine() ) ! = null) {                // (3)
          System.out.println(line);                              // (4)
         }
   input.close();                                                // (5)
  }
}
```

The initial set up and definition of `main` is exactly as for the `PrintWriter`, the new lines are:

(1) Create a `FileReader` connected to the input file and a `BufferedReader` called `input` "on-top" (which implements the simpler interface).

If the input file does not exist, or cannot be read, then an `IOException` results and the program will exit with a suitable message. (See advanced sections of books on `catch`ing exceptions.).

(2) Create an empty `String` to receive the input data.

(3) Use the method `readLine()` to read the next line from `input`, returning a `String`.

If `readLine()` fail to read a line, typically due to running off the end of the file, it will return the `null` String. This loop therefore runs until there are no more lines to be read from the input file. Note the order of the brackets, this forces the statement

```
line = input.readLine()
```

to executed before the conditional test.

(4) Output the `String` we have just read to the terminal output.

(5) Close the input file, not essential but good programming practice.

So again as with `PrintWriter`, if the syntax for opening the `BufferReader` is used as a recipe, the reading of lines from a input file is very simple. The real difficulty is in interpreting the input `String`, for example if you want to read `int`s or `double`s into your program. Here the best strategy is to read the input file "line-at-a-time" and then break up the input line extracting `int`s or `double`s inside your program. See the next section for this.

[2] You would logically think that there should be a `PrintReader` class, but there is not one.

13.4.1 Input from the keyboard with `BufferedReader`

Unlike most other languages the current JAVA does not offer a simple "read string from keyboard"[3], but this can be implemented by `BufferedReader` as follows, which prompts the user for an input, reads the line and then outputs it back to the screen

```
import java.io.*;
public class ReaderTest {
   public static void main(String args[]) throws IOException {
   BufferedReader keyBoard = new BufferedReader(                // (1)
         new InputStreamReader(System.in));
   String line;
   while (true) {
       System.out.print("Type in a line : ");                   // (2)
       line = keyBoard.readLine();                              // (3)
       System.out.println("The typed line was : " + line);      // (4)
      }
    }
}
```

Again the initial setup and `main` is identical to above, the key lines are.

(1) Open a `BufferedReader` called `keyBoard` that reads from an input STREAM connected to `System.in`, which is connected by default to the terminal key board. (Again use as a recipe).

(2) Print a prompt to the terminal. Note the use of the `print()` method. This prints a `String` but *not* a new line.

(3) Read a line from the `keyBoard`. This method will wait until you press RETURN.

(4) Output the line back to the terminal.

Note this program will loop infinitely, use CTRL-C to break out.

13.5 Breaking up input lines with `Scanner`

Most scientific programmers want to read numerical data into their programs, for example to read in columns of numbers into arrays which the program then processes. This is always a tricky problem especially if the exact format of the input lines is not known exactly so you have to "hunt" for the start of the numbers which may be surrounded with stray WHITE SPACE characters, such as SPACE, TAB etc. The problem is called TOKENIZING a string and there is a very useful class with the JAVA `util` package that does almost all the work for you. The following example does exactly what we want,

```
import java.util.*;                                     // (1)
< ---     usual start of program   ----- >
String line = "Hello World and Welcome";                // (2)
```

[3]This is why you have been using the `Display` class for you initial programs.

```
Scanner tokens = new Scanner(line);                    // (3)

while(tokens.hasNext()) {                              // (4)
   String s = tokens.next();                           // (5)
   System.out.println("Token is : " + s);              // (6)
}
```

Looking at the important lines, we have

(1) Import the `java.util` package that contains the `Scanner` class.

(2) Create a long `String` with multiple words.

(3) Create a `Scanner` object taking the long `line` as a parameter.

(4) Read the tokens formed. The method `hasNext()` is `true` if there is a token available for reading.

(5) Read the next available token. The method `next()` returns the next available token as a `String`.

(6) Print the returned token to the screen.

This piece of code will print out the *four* tokens, being "`Hello`", "`World`", "`and`", "`Welcome`".

For a working example of this, see the `KeyBoardInput` example from the Examples page. This program prompts for an input line, reads it into a `String`, tokenizes it, and then prints the token back out the the screen. Run this program with various input lines and see what it does. By default `Scanner` assumes that there is a space between each token, this can however be altered to a wide range of characters or logical combination of patterns, see JAVA documentation for details.

This may look overly complex, but illustrates one of the main concepts in object oriented programming. That is use of pre-defines, and hopefully, well tested objects to do the "work". You do not need to know what is inside `Scanner` you just need to know how to construct it and access the tokens it generates. The more advanced your programming gets the more your programs consist of objects and methods to access them.

`Scanner` is the new *easy to use* tokenizer object introduced to JAVA 5, the previous one being `StringTokenizer`. `Scanner` has a simpler syntax, supports direct reading of `int`, `double`, etc., and will also read *direct* from input files or even `System.in`. This last feature makes the syntax of reading from files simpler, but I personally feel, makes debugging and error recovery much more difficult. I suggest that novice programmers still use the strategy of,

1. Open file for reading
2. Read a line into a `String`.
3. Analyse the contents of the `String`
4. Try and read the next line...

and leaving anything more cleaver until they have mastered this scheme.

13.5.1 Reading `int`s and `double`s

We have seen above how to break a long line up into tokens, but when you read in data what you will almost always want to do is to is set `int` or `double` values. `Scanner` provides method to return token as `int` or `double` using the `nextInt()` and `nextDouble` methods[4], so if a `String` called `line` contains two `doubles`, we can read them with,

```
Scanner scan = new Scanner(line);
double x = scan.nextDouble();
double y = scan.nextDouble();
```

There are also very useful `boolean` *companion* methods with allow you to *look ahead*, to see if the next token can be read as specified. For `int` and `double` these are,

`boolean hasNextInt()` and `boolean hasNextDouble()`

which will return `true` *if* the next token is a valid `int` or `double` respectively.

So putting this all together leads to the more complex example of `PlotGraphFromFile` from the examples below, which uses the `Display` class to ask for an input file containing two columns of numbers. The file is read in line at a time, tokenized and parsed into `doubles` and then then plotted using `SimpleGraph`. This example also ignore blank lines and lines that start with the `#` character which, by convention, is used to denote comment lines in a data file. You can experiment with either data produced by the `CosPrinter` program or data typed into using the `emacs` editor.

Note: This program will fail in a most un-graceful way if the data format is wrong, a data value is missing or the file contains stray, unexpected, characters. Catching and dealing with such errors is difficult and well beyond a simple programming course. However using the above strategy of reading lines, then analysing them is the correct start. For example it would be easy to put a check that we get two tokens form each line, and if not, print out a useful message saying on which line of the input data file the error occurred. Testing and checking input is vital to getting a program to work correctly and robustly.

13.5.2 Converting `String`s to `int`s and `double`s

In many applications you need to convert a single `String` into an `int` or a `double`. Clearly this can be done with `Scanner`, but there are two simpler method, being `Integer.parseInt()` and `Double.parseDouble()` respectively, that both take `String`s as their parameter and return the `int` or `double` value respectively. So in the code example below,

```
String intString = "-45";
String doubleString = "45.7643e-3";

int iValue = Integer.parseInt(intString);
double dValue = Double.parseDouble(doubleString);
```

then *iValue* will be set to -45 and `dValue` will be set to 0.0457643.

[4]There are also method to read `Float`, `Byte` etc.

Unfortunately the `parse` methods also fail if there are stray leading or trailing WHITE SPACE characters in the `String`[5]. To prevent this problem it is better to use the `String` method `trim()` which returns a `String` with any leading or trailing WHITE SPACE characters removed. So a better, and more robust, use of the `parse` methods is

```
String intString = " -45";
String doubleString = "45.7643e-3 ";

int iValue = Integer.parseInt(intString.trim());
double dValue = Double.parseDouble(doubleString.trim());
```

which, correctly, ignores leading or trailing spaces.

13.6 Summary and Additional Features

JAVA has possibly the most complete and thus complex input/output system of any computer language allowing almost unlimited tailoring and optimisation. Here we have seen the very basics of file input/output and how to read, tokenize and parse simple `int` and `double` inputs. For many simple programs this is really all you will need. The two useful advanced areas worth looking at are:

1. Binary data input/output were we read and write direct binary representations of numbers, strings etc. This is *much* more efficient that character input/output and is used where there is large amounts of data, for example graphics files, images, sound files, movies etc. In most cases there are high-level classes to handle various file types either as part of JAVA or add-on packages, the most useful being `java.imageio` with handles images in `jpeg`, `png` or `bmp` formats.

2. Pipe input/output where the reads/writes at to/from the write/reads of other programs. This allows data to be transferred between programs, one program to control another or data to be sent direct to system utilities, for example being able to print data directly to system printer by "piping" your output directly to `lpr`, the standard system print command.

In addition there is a list of less common input/output areas, including "random access files", mainly used in data-base system, binary reading and writing complex objects where efficiency is essential and network files access for example accessing a WEB page on a remote machine.

Examples

Source code for the following on-line examples are available,

- Read line from keyboard and tokenize with `Scanner` KeyBoardInput.
- Write $x, \cos(x)$ pairs out to a specified file with one pair on each line CosPrinter
- Complete program to read in data pairs from a file and plot using `SimpleGraph` PlotGraphFromFile and some test data Here.

[5] This is the only place I know in JAVA where extra spaces do really matter.