

# 21

# Text Files

## 1 Creating text files: `PrintWriter`

We have employed text files by redirecting the standard input and output, but this way of handling files has limitations. For example, it prevents the program outputting to both a file and the screen, and it doesn't allow us to create more than one file. Moreover, the program must rely on the user re-directing input and output. We present ways of creating and reading files under program control. For the moment we will be creating *text files*, by which is meant that the files are composed of normal text such that they can be read using a text editor. In fact a text file created by a program is indistinguishable from one created with a text editor. Later we will see alternatives to text files. Handling files by redirecting the standard input and output remains valid; the techniques below are additional tools.

Handling text files in Java requires knowledge of two classes: class `PrintWriter` is used for creating and writing to text files, and class `Scanner` is used for reading them. We create a text file by instantiating **`PrintWriter`**:

`PrintWriter(String)` throws `IOException`

**`new PrintWriter(s)`** creates a file called `s` on the disk. For example, we create a text file called `data.txt` as follows:

```
PrintWriter myFile = new PrintWriter("data.txt")
```

Note that the creation of a text file may generate an I/O exception which the programmer must be prepared to deal with (see the example program below). When the `PrintWriter` object is

created, an empty file is simultaneously created in the same folder (directory) as your program (i.e. in the same directory as the .class file). The name of the file in the above example is `data.txt`, but you can choose any name as long as it is permitted by the operating system (most operating systems have rules, such as forbidding names containing '/' or spaces). The name doesn't have to have a .txt suffix or any suffix at all, unless required by the operating system. However, in some systems it is common to include a suffix such as .txt to indicate that the file is a text file. If the file is to be created in some other folder, the *path name* of the file should be supplied. A typical path name in Windows is `c:\mybox\myjava\data.txt` – this indicates the file called `data.txt` is located in a folder called `myjava`, and this folder is in turn contained within folder `mybox` located on drive `c`. In Java, it is always acceptable to use forward slashes in place of backslashes in path names, as in `c:/mybox/myjava/data.txt`. Indeed it is more convenient to do so because Java requires that backslashes be duplicated when they are included in strings, as in `"c:\\mybox\\myjava\\data.txt"`. In Unix, a typical path name is `/users/students/smith/mybox/myjava/data.txt`. Here, `/users/students/smith` identifies the user's home directory, and file `data.txt` occurs in directory `myjava` which lies within directory `mybox` which in turn lies in the home directory. The path name may be given relative to the folder in which the program runs, by starting the path with a folder assumed to be in the current folder. For example, under Windows the path `temp\data.txt` refers to `data.txt` in folder `temp` which is taken to reside in the same folder as the executing program (e.g. if the program resides in folder `c:\mybox\myjava` then the full path name of `temp\data.txt` is `c:\mybox\myjava\temp\data.txt`). The following example illustrates the use of a path name:

```
PrintWriter myFile = new PrintWriter("c:/mybox/myjava/data.txt")
```

In informal discussion we often talk about a file on disk by referring to the Java object that is created with it. Given the preceding declaration, for example, we may use `myFile` in place of `data.txt` to refer to the file.

Once the file is created, you can write text to it using the following `PrintWriter` methods

```
void print(String)
void println(String)
PrintWriter printf(String, Object...)
```

**f.print(s)** and **f.println(s)** each append `s` to text file `f`, and in the case of `println()` terminates the line. In Windows, lines are terminated by a carriage return character ('\r') followed by a newline character ('\n'), whereas in Unix just a single carriage return character is used. For example, if integer variable `n` contains 3 then the statement

```
myFile.println("Result = " + n);
```

appends `"Result = 3"` to file `myFile` and terminates the line. There are versions of `print()` and `println()` for all the basic types:

```
void print(int)
```

```
void println(int)
void print(double)
void println(double)
...
```

**f.printf()** is used just like `System.out.printf()`, except that the output is directed to the file identified by `f`. For example, `myFile.printf("Result = %d days", n)` appends "Result = 3 days" to file `myFile` assuming variable `n` contains 3. The value returned by `printf()` is not of interest and we nearly always ignore it (i.e. we treat the return type as `void`). Its parameters (other than the first) are declared to be of type `Object` which for our purposes just means that any type of argument can be supplied in practice.

When you have finished writing to the file you must invoke the following method in `PrintWriter`:

```
void close() throws IOException
```

**f.close()** *closes* text file `f`. If you do not close the file, data may be lost, so don't forget it! Once you close a file you may not write to it further unless you "open" it again (see below).

Java's classes are organised into *packages*. Classes concerned with files, such as `PrintWriter`, are located in a package called `java.io`. The full name of a class in the Java library includes the package name as a prefix followed by a period followed by the name as it occurs in the class definition. For example, the full name of `PrintWriter` is `java.io.PrintWriter`. You must use the full name in your program unless you make the class available by *importing* it. To import class `PrintWriter`, say, you place the following importation statement before the definition of the class that uses it:

```
import java.io.PrintWriter;
```

Thereafter, the program can use the short name `PrintWriter` in place of the long-winded `java.io.PrintWriter`. Alternatively (and more conveniently) you can import all the classes in package `java.io` by writing:

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

and this is what we usually do. Not all packages must have their classes explicitly imported. We will mention it when importation is necessary.

### *Example 1: passes and failures file*

As an example, the program below reads a sequence of student records from the keyboard and creates two text files – a *passes* file and a *failures* file. Each student record occurs on a line and consists of forename, surname, and a percentage mark. A typical input is

```
Bill Smith 69
Jill Wright 43
Anne Butler 89
Max Wallace 38
```

The failures file (which we'll call `bad.txt`) is to contain the names of all students whose mark is less than 45, and the passes file (which we'll call `good.txt`) is to contain the names of all students whose mark is at least 45. We provide two versions of the program, one which handles I/O exceptions, and one which ignores them. The first version follows. Observe that a single exception handler is provided rather than one for each file operation. This is the preferred style to avoid cluttering the code with try-catch blocks.

```
import java.io.*;
class TextWrite {
    public static void main(String[] args) {
        try {
            final int passMark = 45;
            PrintWriter passes = new PrintWriter("good.txt");
            PrintWriter failures = new PrintWriter("bad.txt");
            while (!Console.EndOfFile()) {
                String forename = Console.readToken();
                String surname = Console.readToken();
                int mark = Console.readInt();
                if (mark < passMark)
                    failures.println(forename + " " + surname);
                else
                    passes.println(forename + " " + surname);
            }
            passes.close(); failures.close();
        }
        catch(IOException e) {
            System.out.println("File handling failure!");
            e.printStackTrace();
        }
    }
}
```

In the second version, we signal that no handler for I/O exceptions is provided by including `throws IOException` in the header for `main()`:

```
import java.io.*;
class TextWrite {
    public static void main(String[] args) throws IOException {
        final int passMark = 45;
        PrintWriter passes = new PrintWriter("good.txt");
        PrintWriter failures = new PrintWriter("bad.txt");
        while (!Console.EndOfFile()) {
            String forename = Console.readToken();
            String surname = Console.readToken();
```

```

        int mark = Console.readInt();
        if (mark < passMark)
            failures.println(forename + " " + surname);
        else
            passes.println(forename + " " + surname);
    }
    passes.close(); failures.close();
}
}

```

In this case, any exceptions will be handled by the Java run-time system. Files created using `PrintWriter` are just regular text files whose contents may be examined by opening them in a text editor (such as `WordPad` or `Notepad` in Windows).

The names of the files in the program above (`bad.txt` and `good.txt`) have been built into the program. If we prefer, we could allow the user to give the files names of his or her choosing, for example by entering them as command-line arguments:

```
java TextWrite pass.txt fail.txt
```

(note there is no `>` in the command line because we are not engaging in redirecting the standard output here – each of `pass.txt` and `fail.txt` is a string argument to method `main()` of class `TextWrite`). In that case, the `PrintWriter` declarations in the programs should be

```

PrintWriter passes = new PrintWriter(args[0]);
PrintWriter failures = new PrintWriter(args[1]);

```

It is still possible to take input from a text file (`students.txt`, say) by re-directing the standard input:

```
java TextWrite good.txt bad.txt < students.txt
```

Alternatively, the user could be prompted to key in the names of the files during execution:

```

System.out.print("Enter name of passes file: ");
String passFile = Console.readString();
System.out.print("Enter name of failures file: ");
String failFile = Console.readString();
PrintWriter passes = new PrintWriter(passFile);
PrintWriter failures = new PrintWriter(failFile);

```

At the first prompt, the user should enter, say, `good.txt`, and at the second `bad.txt`.

### Appending to existing text files

You may append text to an already existing text file. You “open” the file (i.e. make it available to the program) by creating a `PrintWriter` object as before, but this time wrapping the filename in some additional text:

```
PrintWriter(new FileWriter(String, boolean)) throws IOException
```

`new PrintWriter(new FileWriter(s,true))` makes the text file named `s` on the disk available for output, such that all data will be written at the end of the file right from the start. The file will normally already exist, but if not then an empty file with the given name will be created. As an example:

```
PrintWriter myFile = new PrintWriter(new FileWriter("data.txt", true))
```

opens file `data.txt` for appending. Although class `FileWriter` is used as an intermediary in the constructor; no further knowledge of it is needed in order to use class `PrintWriter`.

## 2 Reading from text files: Scanner

Text files, whether produced by a program or with a text editor, can be read by a program using class `Scanner`, part of the `java.util` package. We “open” a file for reading by creating a `Scanner` object:

```
Scanner(new File(String)) throws FileNotFoundException
```

`new Scanner(new File(s))` makes the file called `s` on the disk available for reading. Class `File` has no role to play other than as an intermediary in the construction of a `Scanner` object, and we need not ourselves with it further. The file named `s` should exist in the same folder as the executable program, unless a path name is given. It doesn’t matter how the file was created – it could have been created by a text editor, or a program using class `PrintWriter`, or a program using redirection of the standard output, or by any other means. The following is an example of using `Scanner`:

```
Scanner myFile = new Scanner(new File ("data.txt"));
```

`Scanner` provides input via the methods below. In each case, reading begins at the start of the file and advances with each read operation until the end of the file.

```
String nextLine()  
String next()  
int nextInt()  
long nextLong()  
double nextDouble()  
boolean nextBoolean()
```

`myFile.nextLine()` returns the next line from file `myFile` (without the trailing end-of-line delimiter). If the file is currently positioned in the middle of a line, the remainder of the line is returned. It is an error if there is no more input. `myFile.next()` skips whitespace until it finds a token and then reads and returns the token (reading just to the end of the token, and no more). *Whitespace* means all characters that normally separate words, such as spaces and end-of-line characters. A *token* is a maximal sequence of characters other than whitespace. A *token* is just another name for a “word”, but we prefer *token* to indicate that the word need not consist of letters only but could include, for example, digits and punctuation characters. For example, the tokens in “30/20 equals 1.5” are “30/20”, “equals”, and “1.5”. Formally, a token

is a maximal sequence of characters not containing a separator, where a separator is a space, a tab character, or any end-of-line character (i.e. a newline or carriage-return). Tokens may be interspersed with any number of separators; for example the tokens in “ 30/20 equals 1.5 ” are the same as those in “30/20 equals 1.5”. `myFile.nextInt()` skips whitespace until it finds a token and then reads and returns the token as a value of type `int`. It is an error if the token does not represent an integer. `nextLong()`, `nextDouble()`, and `nextBoolean()` behave similarly.

A word of caution: The above methods other than `nextLine()` do not read any end-of-line delimiters. So each line should be read either with a single `nextLine()` or with some sequence of read actions ending with a `nextLine()`.

`Scanner` provides the following methods for detecting what the next read operation will yield:

```
boolean hasNextLine()
boolean hasNext()
boolean hasNextInt()
boolean hasNextLong()
boolean hasNextDouble()
boolean hasNextBoolean()
```

`myFile.hasNextLine()` indicates whether there is another line in `myFile`, and `myFile.hasNext()` indicates whether there is another token. `myFile.hasNextInt()` indicates whether the next token can be interpreted as a value of type `int`. `hasNextLong()`, `hasNextDouble()`, and `hasNextBoolean()` behave similarly.

When you have finished reading the file you should close it by invoking the following method in `Scanner`:

```
void close()
```

`myFile.close()` closes file `myFile`. No harm is done if a file open for reading is not closed, but open files consume quite a chunk of memory and so it pays not to leave them open unnecessarily.

Closing a file is also useful if you want to read it a second time. Just close the file and create a new `Scanner` object associated with the file name.

### *Example 1: counting words*

The following example program counts the number of words in a text file, where the name of the file is supplied as a command-line argument. For example, the following command line counts the number of words in a file called `source.txt`:

```
java CountWords source.txt
```

Note again that `source.txt` is not preceded by `<` – it is no more than a string supplied as a

parameter to `main()`.

```
import java.io.*;
import java.util.*;
class CountWords {
    public static void main(String[] args) {
        try {
            Scanner in = new Scanner(new File(args[0]));
            int numWords = 0; // number of words
            while (in.hasNext()) {
                numWords++; in.next();
            }
            in.close();
            System.out.println(numWords + " words");
        }
        catch(IOException e) {
            System.out.println("File unreadable");
            e.printStackTrace(); //optional, for additional info
        }
    }
}
```

`java.util.*` is imported so that class `Scanner` is available, and `java.io.*` is imported so that i/o classes such as `File` are available.

### 3 Reading from text files: `ConsoleReader`

Instead of `Scanner` we can use **`ConsoleReader`** to handle input from text files. It has the advantage that it operates identically to `Console`, and the disadvantage that it is not part of the standard Java library. It is best to use `Scanner` when it's convenient to do so, but `Scanner` has the disadvantage that it provides the input as either tokens or lines while `ConsoleReader` can additionally supply the input character by character. You can get a copy of `ConsoleReader` from the home web page.

`ConsoleReader` behaves identically to `Console`, except that you have to open the text file before you start reading from it, and you should close it when you are finished. To open a text file for reading you construct an object of type `ConsoleReader`:

```
ConsoleReader(String)
```

**`new ConsoleReader(s)`** makes the text file named `s` on the disk available for reading (as well as creating a `ConsoleReader` object). For example, the following opens a text file called `data.txt`:

```
ConsoleReader myInput = new ConsoleReader("data.txt");
```

`ConsoleReader` handles all I/O exceptions internally, so no try-catch blocks are needed. For every method of `Console`, there is a similar one in `ConsoleReader` – just replace the prefix `Console` with a reference to a `ConsoleReader` object. For example, if the first data item in file `data.txt` just opened above is an integer, it can be read by executing

```
int n = myInput.readInt();
```

`ConsoleReader` provides the following method:

```
void close()
```

**f.close()** closes file `f`. No harm is done if the file is not closed, but open files consumes memory space so it is good housekeeping not to keep too many files open.

### *Example 1: counting words*

The following example program counts the number of words in a text file, where the name of the file is supplied as a command-line argument. For example, the following command line counts the number of words in a file called `source.txt`:

```
java CountWords source.txt
```

The program uses the fact that `readToken()` returns `null` if there is no more data in the file.

```
class CountWords {
    public static void main(String[] args) {
        ConsoleReader in = new ConsoleReader(args[0]);
        int numWords = 0; // number of words
        String w = in.readToken();
        while (w != null) {
            numWords++;
            w = in.readToken();
        }
        in.close();
        System.out.println(numWords + " words");
    }
}
```

## 4 Reading text files from the world-wide web OPTIONAL

Every file on the world-wide web has a unique identification called an URL (which stands for *Uniform Resource Locator*). A typical URL is `http://www.tug.org/tex-ptr-faq` which identifies a file called `tex-ptr-faq` residing on a web site known as `www.tug.org`. You will be familiar with URL's from web browsers such as Internet Explorer or Firefox. In fact, each page you download from the web is just a text file which contains textual information to be displayed intermixed with textual instructions on how the browser should display it. `Scanner` can be used to read a text file identified by an URL, using the following constructor

```
Scanner(
    new InputStreamReader(
        (new URL(String)).openStream())) throws IOException
```

The string argument is the URL identifying the text file. The constructor uses classes **InputStreamReader** and **URL** as intermediaries, and although they look frightening, no understanding of them is needed. Once the `Scanner` object is created it is used just as though it was associated with a text file residing on your machine. `InputStreamReader` is located in package `java.io`, and `URL` is located in package `java.net` and so programs that use them must include the appropriate import statements.

### *Example 1: Printing a file from the web*

As an example, the following program prints out file `http://moodle.dcu.ie`.

```
import java.io.*;
import java.util.*;
import java.net.*;
class ReadFromUrl {
    public static void main(String[] args) {
        try {
            Scanner in = new Scanner(
                new InputStreamReader(
                    (new URL("http://moodle.dcu.ie")).openStream()));
            while (in.hasNextLine()) {
                String s = in.nextLine();
                System.out.println(s);
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

If you are executing the program from a machine on a local network, your program may be refused permission to access external files without routing the request through a *proxy server*, i.e. a machine which manages and regulates external web traffic to ensure security. The web proxy in the School of Computing, for example, is called `wwwproxy.computing.dcu.ie`. Every server provides a range of services, each service being identified by a *port number*. The port number `wwwproxy.computing.dcu.ie` uses for dealing with web access requests is 8000. You can inform Java of all this by including the following in your program

```
System.setProperty("http.proxyHost", "wwwproxy.computing.dcu.ie");
System.setProperty("http.proxyPort", "8000");
```

Once these statements have been executed, subsequent web requests will be routed through the web proxy allowing your program external access to the web.

## 6 Parsing text: Scanner

The `Scanner` class can be used to extract the tokens from a string: just create an instance of `Scanner` with the string as argument to the constructor:

```
Scanner(String)
```

`new Scanner(s)` creates an instance of `Scanner` in which the tokens and lines read are taken from `s`. For example:

```
Scanner t = new Scanner(" some string this");
System.out.print( t.next() + t.next());
```

causes `somestring` to be displayed on the screen. When using `Scanner` in this way, there is no need to invoke `close()` at the end.

## 7 Example: student records

The following case study illustrates the use of text files where each item in the file is composed of a few sub-items. We make two programs, one of which creates a text file of student records, and the other of which queries the file. The program to create the student file will be invoked by the command

```
java CreateStudents students.txt
```

This creates a text file of students (here called `students.txt`) in which each line contains the name (forename and surname), sex (boolean `true` for male), and exam mark of a single student. The program with some sample students follows. Note that we use `println` (rather than `print`) to write the final item in each student record; this makes the information for each student easily readable when we examine the file.

```
import java.io.*;
class CreateStudents {
    public static void main(String[] args) {
        try {
            // Create a small test file of students
            PrintWriter out = new PrintWriter(args[0]);
            out.print("Jill Jones" + " "); out.print(87+ " "); out.println(false);
            out.print("Michael MacDonald" + " "); out.print(19+ " "); out.println(true);
            out.print("Pete Pineapple" + " "); out.print(65+ " "); out.println(true);
            out.print("Jenny Murphy" + " "); out.print(49+ " "); out.println(false);
            out.close();
        }
    }
}
```

```

        catch (IOException e) {
            System.out.print("Could not create file " + args[0]);
        }
    }
}

```

The querying program displays the names of students whose mark exceeds a value supplied in the command line. For example, the following command displays the names of students whose mark exceeds 60 in a file called `students.txt`:

```
java ListStudents students.txt 60
```

The program follows. Note carefully that after each student is read, it is necessary to read the end-of-line character by invoking `nextLine()`.

```

import java.io.*;
import java.util.*;
class ListStudents {
    public static void main(String[] args) {
        int divMark = Integer.parseInt(args[1]);
        try {
            Scanner in = new Scanner(new File(args[0]));
            while (in.hasNextLine()) {
                String name = in.next() + " " + in.next();
                int mark = in.nextInt();
                boolean isMale = in.nextBoolean();
                in.nextLine(); // remember to read end-of-line
                if (mark >= divMark)
                    System.out.println(name);
            }
            in.close();
        }
        catch (IOException e) {
            System.out.print("Could not access file " + args[0]);
        }
    }
}

```