22 Reading Text Files

1 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 concern ourselves with it further; it resides in package `java.io` and must be imported. 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 (i.e. a newline or carriage-return). 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". 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.

`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. Always choose the appropriate method to detect the end of input; for example, if you are reading the file line by line, use `hasNextLine()`, not `hasNext()`.

The read methods in `Scanner` other than `nextLine()` recognise but do not read an end-of-line character marking the end of a token. This may occasionally make it necessary to read an end-of-line character explicitly using `nextLine()`. Similarly, caution is required when using `hasNextLine()` in a loop to detect an end of file: make sure no end-of-line character is left unread (see example later).

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.*; // for File class
import java.util.*; // for Scanner class
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
    }
  }
}
```

2 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");
   }
```

```
}
```

3 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.

4 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.

5 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]);
            }
        }
    }
```