Programming via Java File I/O

Most real programs must access files at some point. We'll investigate a couple of simple ways to work with files using Java's libraries.

22.1. The `char` type

But first, we will study the **`char`** type, a primitive type built into Java for representing a single character. This is a new primitive type, in addition to the three we've already seen: **`int`**, **`double`**, and **`boolean`**.

A **character** is any single symbol that might appear in a sentence, such as a letter in the English alphabet (*a* or *A*, for example), a single digit (*0*), a punctuation mark (*%*), or a letter in some other alphabet (the Hebrew letter aleph א). We can reference any character value in Java by enclosing it in single quotation marks: `'a'`.

```
char letter = 'a';
```

For beginners, the distinction between **`char`** and `String` can be a little confusing, but with practice it's an easy distinction to make: The **`char`** type is for a single character, while the `String` type represents a sequence of characters (though admittedly the sequence may have just one character). We use single-quotation marks for **`char`**values and double-quotation marks for `String` values.

When we have a variable referencing a **`char`**, we can convert it to a `String` by adding it to the empty String.

```
String word = "" + letter;
```

And when we have a variable referencing a `String`, we can retrieve a single characte from it using its `charAt` method. This method takes a single **`int`** parameter giving the index of the character we want from the `String`, and it returns that **`char`**.

```
char firstLetter = word.charAt(0);
```

In Chapter 8 we saw a program that read a line from the user and displayed that line in reverse. It used the `substring` method to retrieve each individual string, since that is all we had seen. Using the `charAt` method is easier than `substring`, though, when we want to retrieve just a single character.

```
String str = readLine("Type a string to reverse: ");
int toPrint = str.length() - 1;
while(toPrint >= 0) {
    print(str.charAt(toPrint));
    toPrint--;
}
```

Because the **char** type is a primitive type, Java programs can compare **char** values using the == operator, as opposed to the `String` type, which is a class and whose values should be compared using the `equals` method.

```
int count = 0;
for(int i = 0; i < str.length(); i++) {
    if(str.charAt(i) == 'r') count++;
}
```

22.2. Elementary file manipulation

Java includes a number of classes for reading and writing files. Two of the most elementary are `FileReader` and `FileWriter`, both in the `java.io` package.

`FileReader(String filename)`

(Constructor) Constructs an object for retrieving characters from the text file named `filename`, throwing a `FileNotFoundException` if the file could not be opened. Possible reasons for the exception include an absence of a file with the name, a lack of permission for reading the file, or an attempt to read a directory as if it were a regular file.

`int read(char[] buffer)`

Attempts to fill `buffer` with characters from this file,

`FileWriter(String filename)`

(Constructor) Constructs an object for saving characters into a text file named `filename`, throwing a `IOException` if the file could not be opened. If the file already exists, all characters will be deleted and the `FileWriter` will start writing at the beginning of the file. Possible reasons for the exception include a lack of permission for creating or replacing a file of the indicated name.

`void write(char[] buffer, int start, int len)`

Writes `len` characters from `buffer` starting from index `start` of the array. Each invocation writes into the file just after the previously written characters. The method

returning the number of characters placed into the buffer or –1 if the file has been exhausted. Each invocation reads the next characters that haven't yet been read. The method may throw an `IOException` on an error, such as the user unplugging a USB drive while a program is reading a file from it.

`void close()`

Closes this file. This should always be done after completing each file, since often the number of open files is limited in most systems. The method may throw an `IOException`.

may throw an `IOException`.

`void close()`

Closes this file. This should always be done after completing each file, since often the number of open files is limited in most systems. Also, writes will often only be placed into a memory buffer, which doesn't actually get written to disk until the file is closed. The method may throw an `IOException`.

These are very elementary classes that aren't often very convenient to use, but they can be used for simple operations like copying all the characters in one file into another file, as in Figure 22.1.

Figure 22.1: The `FileCopy` program.

```
1  import acm.program.Program;
2  import java.io.FileReader;            import java.io.FileWriter;
3  import java.io.FileNotFoundException; import java.io.IOException;
4
5  public class FileCopy extends Program {
6      public void run() {
7          // open input file 'src' and output file 'dst'
8          FileReader src;
9          try {
10             String srcName = readLine("Source file name? ");
11             src = new FileReader(srcName);
12         } catch(FileNotFoundException e) {
```

```
13              System.out.println("Cannot open source file");
14              return;
15          }
16
17          FileWriter dst;
18          try {
19              String dstName = readLine("Destination file name? ");
20              dst = new FileWriter(dstName);
21          } catch(IOException e) {
22              System.out.println("Cannot open destination file");
23              return;
24          }
25
26          try {
27              // repeatedly fill array from 'src' and save into 'dst'
28              char[] buf = new char[128];
29              while(true) {
30                  int n = src.read(buf);
31                  if(n < 0) break; // end of file reached
32                  dst.write(buf, 0, n);
33              }
34
35              // close files
36              src.close();
37              dst.close();
38          } catch(IOException e) {
39              System.out.println("I/O error: Copy may be incomplete");
40          }
41      }
42  }
```

You can see that we have to use several **`try`** blocks to deal with all the exceptions that can arise in the process of working with files. These **`try`** blocks are required by Java: An `IOException` is a checked exception, so the Java compiler forces the programmer to deal with the exception somehow.

The core of the `FileCopy` program is in lines 29 to 33. Here, we repeatedly fill the buffer `buf` from the source file, letting `n` be the number of characters placed into the buffer. And then we dump that many characters from `buf` into the destination file before repeating. Eventually, the `FileReader` will have exhausted the source file, and so its `read` method

would return –1; at that point, the program breaks out of the loop so that both files can be closed.

22.3. Text file manipulation

In practice, dealing with files using `FileReader` and `FileWriter` is often a pain; we often want to process the file as a sequence of strings rather than as a sequence of characters. Two classes `Scanner` (found in `java.util`) and `PrintWriter` (found in `java.io`) are more helpful.

`Scanner(Reader source)`

(Constructor) Constructs an object for retrieving meaningful chunks of text from the characters given by `source`.

`boolean` `hasNextLine()`

Returns **`true`** if there is a line of text in the source file that hasn't yet been returned by this object.

`String nextLine()`

Returns the next line that hasn't yet been returned from the source file. The character(s) representing the line break at the line's end is not included in the returned line.

`void` `close()`

Closes the source file.

`PrintWriter(Writer dest)`

(Constructor) Constructs an object for placing meaningful chunks of text into `dest`.

`void` `print(String str)`

Writes the characters found in `str` at the end of the destination file.

`void` `println(String str)`

Writes the characters found in `str` at the end of the destination file, followed by the character(s) representing a line break.

`void` `close()`

Closes the destination file.

The `FileSearch` program of Figure 22.2 illustrates these two classes at work.

Figure 22.2: The `FileSearch` program.

```
1  import acm.program.Program;
2  import java.io.FileReader;              import java.io.FileWriter;
3  import java.io.FileNotFoundException; import java.io.IOException;
```

```
4  import java.io.PrintWriter;           import java.util.Scanner;
5
6  public class FileSearch extends Program {
7      public void run() {
8          // open input file 'src' and output file 'dst'
9          Scanner src;
10         try {
11             String srcName = readLine("Source file name? ");
12             src = new Scanner(new FileReader(srcName);
13         } catch(FileNotFoundException e) {
14             System.out.println("Cannot open source file");
15             return;
16         }
17
18         PrintWriter dst;
19         try {
20             String dstName = readLine("Destination file name? ");
21             dst = new PrintWriter(new FileWriter(dstName));
22         } catch(IOException e) {
23             System.out.println("Cannot open destination file");
24             return;
25         }
26
27         // determine what string to search for
28         String toFind = readLine("What text should we search for? ");
29
30         // repeatedly read line from 'src' and, if it contains the
31         // desired string, save it into 'dst'
32         while(src.hasNextLine()) {
33             String line = src.nextLine();
34             if(line.indexOf(toFind) >= 0) dst.println(line);
35         }
36
37         // close files
38         src.close();
39         dst.close();
40     }
41 }
```

Source: http://www.toves.org/books/java/ch22-file/index.html