

# Lab 3

*Out: Feb 10-11, 2008*

## Contents

<b>1</b>	<b>Files</b>	<b>1</b>
1.1	File Formats . . . . .	1
1.2	Buffering . . . . .	2
<b>2</b>	<b>The Java IO API</b>	<b>3</b>
2.1	Opening and closing files . . . . .	3
2.2	Reading . . . . .	4
2.3	Writing . . . . .	4
<b>3</b>	<b>Word Count</b>	<b>4</b>
3.1	Concatenate . . . . .	5
<b>4</b>	<b>Shuffling and Juggling</b>	<b>5</b>
<b>5</b>	<b>On the Road</b>	<b>6</b>

## 1 Files

We have used files to store source code (`.java` files) and compiled code (`.class` files), but until now we have not written a program that itself manipulates files. In this lab, that's what you'll be doing.

### 1.1 File Formats

Broadly speaking, files are stored in one of two formats:

- Textually, in a format that humans can understand.

- In binary, much like the way they are represented in your computer's RAM.

There are advantages to both methods.

Textual formats are human-readable and somewhat redundant at the bit level, so errors are more likely to be apparent. Character-based representations are usually a bit looser in structure and are therefore more flexible and forgiving. However, text files generally have to be accessed **sequentially**. Sequential access is the way we process a linked list; you start at the beginning and read through without skipping around. Files are written to only by *appending*, i.e. adding to the end of the file.

Binary representations are already in a form your computer understands. They're more compact, and easier and faster for a program to read. They tend to be a little more error-sensitive and inflexible, but they also allow for *random access*. "Random" here does not mean governed by probability; it means arbitrary. Random access is the way we use an array; an individual block of data can be accessed directly, without first reading through the blocks before it.

If accessing a file sequentially is like using a cassette tape (remember those?), then using random access is like listening to a CD. You can quickly go between tracks of a CD.

Note that sequentially accessed files are not as flexible as linked lists—you can't insert in the middle, and you can only add to the end. Why are they used? They are a good match for the hardware actually used for storing data (disk drives, these days). However, random access to a single integer, say, in the middle of a file takes much longer than reading the integer. For this reason, in a practical system using random-access files, the units of data that are randomly accessed are themselves quite large, typically thousands of bytes. (For example, a CD track is a fairly long chunk of data, and you can't skip to an arbitrary place within a track.)

In CSCI 180, we will only use text files and sequential access.

## 1.2 Buffering

Starting to write to an I/O device such as a disk drive takes a lot of time. For this reason, disk access is usually *buffered*. When reading from a file, a large amount of data is read in at once, and the extra data stays in a

buffer. When writing to a file, the data often does not go directly to the disk drive (or whatever) but is instead stored in memory until enough data are accumulated to make it worth the system's while, at which point the accumulated data are all written out in one fell swoop (we say the data are *flushed*, a particularly vivid term).

## 2 The Java IO API

This lab will involve using Java's support for reading from and writing to files. You'll probably want to keep the CSCI 180 Java API specification <sup>1</sup> open to the `java.io` section during this part of the lab for reference.

You'll notice, in the upper left-hand frame, a list of packages in Java. In the lower left-hand frame, there is a list of all the classes in the Java libraries. If you click on a package in the upper left-hand frame, only the classes in that package will appear in the lower left-hand frame. This is a nice way to navigate through all of the library documentation. Notice that when you click on a class from this frame, documentation for that class appears in the right-hand frame.

In the right-hand frame, there is an overview of all the packages in Java. Following the links on the page in the right-hand frame is another way to navigate through all of Java's libraries. Notice that when you click on a class from the lower left-hand frame, documentation for that class appears in this frame.

Don't worry if you don't understand all the methods in a class. This is a common occurrence with library code, which is built to have as much functionality as possible).

Here are a few tips for getting started.

### 2.1 Opening and closing files

To open a file, you create a file stream object (such as with a `FileInputStream`), and pass the constructor a string with the name of the file. When the stream object goes out of scope, the file is closed. You can also manually close any

---

<sup>1</sup>On the "documentation" section of the course website there are links both to the official JavaDocs as well as an abbreviated CSCI 180 version. The abbreviated version should have all you need, without anything extra.

stream using its `close()` method.

## 2.2 Reading

Take a look in particular at the `FileReader` and `BufferedReader` classes. `FileReader` is obviously useful because it's designed to read from files, and `BufferedReader` is nice because it allows you to read from the file line-by-line using its `readLine()` method. Any time you want to read anything (a file or user input) line-by-line, you'll want to use a `BufferedReader`.

## 2.3 Writing

Take a look in particular at the `FileWriter` and `BufferedWriter` classes. `FileWriter` is obviously useful because it's designed to write to files, and `BufferedWriter` is nice because it allows you to write to the file without incurring the overhead of a disk access per write. You will need to flush the buffered output stream after you are done writing; you can do this by calling the `flush()` method.

## 3 Word Count

**Task:** Create a new class called `Lab03`. All of the methods you write for this lab will belong to this class.

The Linux command `wc`, for “word count”, will count the number of lines, words, and characters in a file.

**Task:** Write a (`public static`) Java method `wordCount` that takes the name of a file and prints the number of lines, words, and characters in that file to the screen.

**Note:** You'll find the `split` method in the `String` class useful: it splits a string into pieces around a specific character and puts them into a `String` array. It takes one argument, a `String` that marks where you want to divide up this string. So, if you wanted to split a sentence into an array of words, you'd split around a space. For example, after running the code

```
String myString = "This just in! Giant weasel-shaped  
spacecraft sighted from Incan ruins!";
```

```
String[] myArray = myString.split(" ");
```

the `myArray` variable would be a `String` array containing the strings:

```
"This", "just", "in!", "Giant", ...
```

and so on.

### 3.1 Concatenate

When applied to things like files or text, the term “concatenate” means, roughly, to “stick these two things together, one after the other.” You will now write a method capable of concatenating two files.

**Task:** Write a (`public static`) method `concatenate`. It should take as input two `BufferedReader`s, `reader1` and `reader2`, and one `BufferedWriter`, `writer`. It should then write to `writer` the contents of `reader1` followed by the contents of `reader2`. Remember to flush the writer when you’re done.

**Task:** Experiment with `concatenate`. Try to get it to write to the console; then try to get it to write to a file.

**Task:** Now write a (`public static`) method `cat`. It should take as input two filenames (strings): `inFile1`, `inFile2`, and `outFile`. It should use the `concatenate` method to concatenate the contents of `inFile1` and `inFile2` into the file `outFile`.

**Note:** To create a standard plain text file in Eclipse, you can right-click on the package you are currently in and select `New > File`. You can edit this file right in Eclipse. If this file is in the same package/folder as your `Lab03` class, the string that represents its path will be the name of the package, followed by a slash, followed by the name of the file (e.g. `labs/foo`).

## 4 Shuffling and Juggling

Now that you have got the hang of reading from files and writing to writers, we can have some fun. And nobody knows fun quite like the Shuffler.

**Task:** Write a (`public static`) method `shuffle`. It should take as input two `BufferedReader`s, `reader1` and `reader2`, and one `BufferedWriter`,

`writer`. It should then write to `writer` alternating lines from the two readers. The method should stop writing as soon as it reaches the end of either reader. Remember to flush the writer when you're done.

**Task:** Write a (`public static`) method `juggle`. It should take as input two `Strings`, `str1` and `str2`, and one `BufferedWriter`, `writer`. It should then write to `writer` alternating *words* from the two strings. As before, if one of the strings has more words than the other, just ignore the extra words. Remember to flush the writer when you're done.

## 5 On the Road

Surely, the Shuffler is tons of fun.<sup>2</sup> But here in CSCI 180, we don't waste your time making you write mere *toys*. Shuffling is in fact an important tool employed by one of the most dynamic and revolutionary industries making waves through today's digital universe; namely, computer-enhanced beat poetry.

In Java, the user input to and from the console travels on streams just like the data we sent to and from files in earlier parts of the lab. You've used these streams before. `System.out` is the console output stream. Whenever you use `System.out.println`, you're sending a string over the `System.out` stream. There's also a `System.in` input stream. Thus, we can create a reader (say, an `InputStreamReader`) that reads from `System.in`. Then, if we create a `BufferedReader` with this `InputStreamReader`, we'll be able to call `readLine` to get a line of user input!

For the final part of this lab, you're going to use `shuffle` to create a beat poetry tool.

**Task:** Write a (`public static`) method `poeticize`. It should take as input two strings, `inFile` and `outFile`. A beat poem is constructed by alternating the lines of `inFile` with lines of user input. Therefore, this method should repeatedly prompt the user for lines of input from the console. Your finished beat poem should be written to the output file specified (allowing you to save your improvised poem for posterity).

**Note:** To tell the console that you're done sending input, you must send it an EOF (end of file), by pressing `Ctrl-D`.

---

<sup>2</sup>TONS of fun.

**CSCI0180**

Lab 3

**Feb 10-11, 2008**

You can test your methods with the files in the `‘/course/cs018/src/lab03’` directory.

Feel free to improve on your program if you have any extra time in lab.