Computer Science 523
Advanced Programming
The College of Saint Rose
Summer 2014

Topic Notes: File I/O

Nearly all computer programs need to do some input and output (I/O) to perform anything useful.

To this point, whenever we have needed input for our programs, our examples gotten it from the keyboard (whether it's a Java `Scanner` using `System.in` or a Java `JOptionPane` dialog box). If we want to run the program again on the same inputs, we need to type in all of the inputs again.

Output has always been sent to a console window or a dialog box. Once we exit our program, unless we have done a copy and paste to save the output, it is lost.

This is another restriction we need to overcome. A very common mechanism for doing this is to take input from and place output in *files*. As you certainly as well aware, reading and writing files is something the programs you use regularly do all the time.

One complication we will see is that we will want to deal with *plain text files*. These are files that consist of nothing but a series of characters (letters, numbers, punctuation). Many of the files we deal with every day (Word documents, web documents, PDF documents) are specially-formatted files that contain much more information that just the text they contain, often in very cryptic and complex formats. Java is capable of dealing with a wide variety of file formats, but we want to concentrate for now only on plain text.

When we wish to create, view, or modify plain text files (outside of our Java programs) at least, we will need to make sure we use an application that can do so properly.

In Windows, you will want to use Notepad. This is a plain text editor that should do just what we want when manipulating plain text files. It should be pretty straightforward to use.

If you are on a Mac, you will want to use the TextEdit application. When you open it, make sure you are in "Plain Text Mode" by looking in the Format menu. If you see an entry labeled "Make Rich Text", then you're already in plain text mode. If you see "Make Plain Text", choose that option to enter into plain text mode.

Java File Output

Writing data to a file in Java is, fortunately, fairly similar to the mechanism we use to print to the screen.

See Example: HelloWorldFile

See the comments in that file for details on what we need to do. In summary:

- 3 new `import` statements: `java.io.File`, `java.io.IOException`, and `java.io.PrintWrite`

- Add a `throws IOException` at at the end of the `main` method signature before its `{`.
- Construct a `PrintWriter` object, passing to the constructor a `new File`, to which you pass the name of the file.
- Print text to the file using `print` and `println` method calls on the `PrintWriter`. These work the same as `System.out.print/println` except they produce file output.
- Call the `PrintWriter`'s `close` method when finished.

We can use this mechanism to store the results of any program for use later. For example, let's write a program that reads in a series of numbers and creates a file with those numbers followed by the area of a circle with that number as its radius.

See Example: CircleAreas

Note that we can use the `printf` variant for output here as well, giving a capability for formatted output. Here, the `%.2f` format specifier indicates that two digits should be printed after the decimal point for the numbers.

Java File Input

As was the case with output, using files for input to our Java programs is also quite similar to what we have been doing when reading from the keyboard. In fact, we will continue to use Java's `Scanner`.

See Example: AddNumbersFromFile

Rather than constructing our `Scanner` with a parameter of `System.in`, we pass a `File`. If that file exists, it will be opened for reading, and we can use the `Scanner` just as we used the `Scanner`s that read from the keyboard.

We saw with keyboard input that a sentinel value can help us to stop a loop that reads values. That technique also works with file `Scanner`s:

See Example: AddNumbersFromFileSenintel

But we can do better than that. When reading from a file, we can continue reading until we encounter the end of the file.

See Example: AddNumbersFromFileAll

We are not restricted to reading numbers. Here is a program that counts the number of words in a file:

See Example: WordCount

Here, we use the `next` method to read the input file word by word. We can tell there are no more words to read when the `hasNext` method returns `false`.

There is no reason to restrict to only file input or only file output.

See Example: Shout

This example uses both an input file and an output file. We read in an input file, line by line, and write each line back out to an output file. However, before we write each line to the output, we call the `String`'s `toUpperCase` method to convert all of the letters to upper case.

There is also a `toLowerCase` method we can use if we'd like.