Introduction to Programming in Java: Lab 9 Reading From and Writing To Files

In this lab, you will learn how to read from and write to files. In the process, you will practice a little more with `JFrame`s and loops.

All the file reading uses classes defined in the package `java.io` so remember to import it.

Task 1. Assigning Lab Partners

If you wish to have a lab partner (and you are strongly encouraged to do so), see your lab instructor. You may keep your lab partner from last week, but you do not need to.

Reminder About The Rules For The Lab

If You Do Not Have a Lab Partner

You are to do the tasks listed below. The tasks require you to write a class that contains specific methods. You need to complete all the methods as described. If you fail to complete all the tasks by the end of lab, you may continue to work on them, but you must complete the lab prior to the deadline posted.

If You Have a Lab Partner

You will play two different roles in the lab. You will either be the *driver* or *navigator*.

- **driver:** The person typing at the keyboard.
- **navigator:** The person watching for mistakes, making suggestions, and thinking ahead.

Important: *The navigator must not touch the keyboard or the mouse.* If the navigator either types or moves the mouse, the navigator will get a zero for the lab.

Choose one of you to begin as the driver. The other will be the navigator, and the lab will tell you when to switch roles.

As in last week's lab, you will write a class that you will submit at the end of lab. You do not have to complete all the tasks by the end of the lab because you are being graded on how well you work as a team. You do have to make a good effort to get full credit.

If you are a novice programmer, your goal is to complete Task 4 and at least get started on Task 5. If we have less than 15 minutes left in lab, and you have not completed Task 4, be sure to get help from the lab assistants.

Task 2. Introductions and Starting DrJava

If you have a lab partner, tell your lab partner your name as well whether you are following either men's or women's NCAA basketball tournaments.

Choose one of you to start as the driver, and launch DrJava.

Task 3. Reading From a File

Basic File Reading

Java provides a few ways to read from a file. The basic way is the `FileReader` class. The following opens a file called `f.txt` for reading, if such a file exists in the appropriate folder:

```
FileReader fr = new FileReader("f.txt");
```

`f.txt` is the name of the file that will be opened. If no folder path is given, DrJava looks in the same folder that it last used to access your Java files. To open a file in a different folder, you should include the full path. For example: `"H:\\MyDocuments\\MyFile.txt"` on Windows machines or `"~/Documents/MyFile.txt"` on Mac or Linux machines. (For Windows, remember to use the control character `\\` when you want the string to contain a single backslash.)

Try it now. Replace `f.txt` in the line above with the name and/or path of a real text file on your system. You can use a Java file or a file you create with a text editor such as the DrJava editor, TextEdit on Mac or Notepad on Windows. Do not use a Word file because it uses proprietary character representations and you will have a hard time reading the results.

Now you can read from the file using the `read` method of `FileReader`. The `read` method is overloaded, and the basic form is to read one character at a time:

```
fr.read()
```

Try this a few times so you can see the first few characters of your file. Note that the return type of `read` is `int`. How do you see the character representation?

Buffered File Reading

Java provides several classes that we can "wrap around" a basic reader class to provide additional functionality. One of those classes is called a `BufferedReader`. `BufferedReader` provides a method called `readLine` that lets us read in an entire line of a file at a time, instead of one character at a time.

To create a `BufferedReader` object, you must first have a `Reader` object (and `FileReader` extends `Reader` and so "is-a" `Reader`).

```
BufferedReader br = new BufferedReader(fr);
```

or, we can combine the two steps:

```
BufferedReader br = new BufferedReader(new FileReader("f.txt"));
```

To read a line, use `br.readLine()`. When there is no more input, `br.readLine()` returns `null`. Test this a couple times so you can see how it works.

Handling Exceptions

An *exception* is an object that indicates that some unusual condition, such as an error, occurred during your program execution. Up to now, we have ignored exceptions because the type of exceptions we have seen, such as `NullPointerException` were *unchecked exceptions*, and your code does not need to explicitly state how it will deal with an unchecked exception.

Most of the methods of `FileReader` and `BufferedReader` can throw an `IOException`, and `IOException` is a *checked* exception. That means we must explicitly state how we will deal with the exception, if it occurs. There are two things you can do with an exception. You can *catch* it (write code that is executed in the event an exception occurs) or you can *throw* it (pass it on).

For this lab, you will throw the exception (pass it on). To pass on the exception, you must add `throws IOException` to the end of any method header that uses a method from one of the `java.io` classes. The `throws IOException` has two purposes. First, it indicates that the method will throw this exception, if it occurs, and second, it warns any method that will use this method that an unchecked exception could be thrown.

Task 4. Displaying a File

Create a class called `FileStuff`.

Create a `static void` method called `fileDisplay` that takes a single `String` as input. The input `String` is the name of a text file whose contents will be displayed on the screen. You will using the `FileReader` constructor and the `BufferedReader` `readLine` method, and both can throw an `IOException` so you must explicitly deal with it:

```
public static void fileDisplay(parameter list) throws IOException {
```

In the method, create a `JFrame` object, and have the title of the `JFrame` be the name of the text file. Create a `JTextArea` object with 40 rows and 80 columns and place it in the center of the `JFrame`. The method should then open a `BufferedReader` for the file specified by the parameter and display the contents of the file in the text area. You should `close` the readers once you are done reading in the file and placing the contents in the `JTextArea`. Make sure you `pack` the `JFrame` and make it visible as the last steps of the method.

For example,

```
FileStuff.fileDisplay("FileStuff.java")
```

should make make a new window showing your class.

Compile it, test it, and fix any errors.

Then, place a JavaDoc comment on the method. Remember to use the `@param` tag for the parameter. There is also a tag to mention the checked exception:

```
* @throws IOException if there is a problem reading from input file.
```

Demonstrate to your instructor or lab assistant that it works.

Task 5. Copying files

Switch roles: The navigator should drive and the driver should navigate.

Writing to a file

Similar to reading from a file, Java provides classes for writing to a file. The base class to write to a file is `FileOutputStream`. As with the `FileReader`, the `FileOutputStream` only writes a byte at a time. A second class, `PrintStream` can wrap around the `FileOutputStream` and provides the ability to write a line at a time.

The following code opens a file for writing; `PrintStream` has methods `print` and `println`, just like with `System.out`:

```
PrintStream p = new PrintStream(new FileOutputStream("file.txt"));
```

So, `p.print("Hello");` will print the string `"Hello"` to the file `file.txt`, and `p.println("Hello")` behaves exactly the same except that it includes a newline character to the end of `"Hello"`.

Your programming task

Create a `static void` method called `fileCopy` that takes two `String` parameters. The parameters are the name of the input file and the name of the output file, and the method should copy the contents of the input file to the output file. As a warning, do not use an important file as the output file; you will lose the contents! Remember that you can use the edit window of DrJava to create any text file and you can open any textfile in the editor. So you can use DrJava (or Notepad or TextEdit) to create your input files and open your output files to verify the contents.

Task 6. Prevent the Overwriting of Existing Files

Switch roles: The navigator should drive and the driver should navigate.

The problem with the `fileCopy` method is that it will overwrite an existing file. To prevent this, modify the `fileCopy` method so that it first creates a `File` object for the output file. Use the `File` object to test whether the output file already exists. If it does, print an error message and do not do the copy.

Optional Task 7. Removing Comments

Switch roles: The navigator should drive and the driver should navigate.

Create a `static void` method called `fileCopyNoComments` that takes two `String` parameters. The method should act the same as the previous `fileCopy` method except that any comments in the file should be removed. For the purpose of this method, we will assume a comment is text inside `(*` and `*)`.

For example, if the input file has a line:

```
We hold these truths to (* should these be obvious?? *) be self-evident,
```

then the output should have

```
We hold these truths to  be self-evident,
```

A comment can be an entire line, inside a line, or it can span multiple lines:

```
Four score (* This can be a tricky comment to remove.
Who even knows what a score is?  Perhaps just saying eighty-seven would
be better *) and seven years ago.
```

and there can be multiple comments in a line.

Task Last. Finishing Up

Submit your `FileStuff.java` file. You can find the submit procedure by clicking on the *Lab 9* link on Blackboard. Remember that both you and your lab partner should submit the file.