Lab Exercise 1
CS 2334

August 27, 2015

Introduction

The goal of this laboratory is to get everyone up to speed in using Eclipse, the Java compiler, Javadoc, archive files and D2L submission. For most, this will require a short period of time to complete. Subsequent labs will occupy the full lab session period.

Learning Objectives

1. Install Java version 8 and Eclipse
2. Compile and execute a Java program in Eclipse
3. Learn and apply basic String-manipulation methods to parse a String
4. Create a class and construct an object of that class
5. Use Javadoc to generate documentation for your code
6. Submit your program and documentation for grading

Proper Academic Conduct

This lab is to be done as individuals. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

Instructions:

1. If you have not already done so:

   - Download and install JDK version 8 at the following address:
     - http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.
   - The use of Java 8 is required for this course. However, if you have already installed a previous version of Java (6 or 7) you will probably be fine. The caveat with using a prior version is that it will be more difficult to get help from the teaching team and there are some small differences from version 8.

     You can check which version of Java you have installed by opening the Terminal (Mac/Linux) or the Command Prompt (Windows) and running the command *java -version*
   - Download and install the latest version of Eclipse IDE for Java Developers (Eclipse Classic works, too) from http://www.eclipse.org/downloads/. If you have Eclipse version 3 or earlier installed, it is recommended that you upgrade to version 4.

2. In Eclipse, create a new Java Project called *lab1*.

   Select *File/New Java Project*

   or

   Select *File/New Project/Java Project*

3. In this project, create a class called `Driver`.

   - Select the lab1 project in the *Package Explorer*
   - Right-click and select *New/Class*
   - Give the class a name and click *Finish*

4. In your `Driver` class, create the `main` method. The `main` method should print the text *Hello, world!* to the screen.

5. Compile and execute your program.

   Left-click on the green *Start* button on the top tool bar.

6. Create a class called *Forecast*. This class will represent forecast information for a day and must have the following properties:

   - It must have an internal String array of size 4 called *info*
   - It must have one constructor:

     ```
     public Forecast(String strg)
     ```

     where input is guaranteed to be a String of the form ω, α, β such that
     - ω is a string consisting of only alphabetical characters, and
     - α, β are strings consisting of only numerical characters.

     For example, *Rainy,42,65* is such a string where ω = *Rainy*, α = *42*, and β = *65*. For this program, we're interpreting ω to be the weather conditions, α to be the day's lowest temperature, and β to be the day's highest temperature.

     This method will store ω in all capital letters in `info[0]`, store α in `info[1]`, store β in `info[2]`, and store the difference $\beta - \alpha$, the day's temperature range, in `info[3]`. Note that this last operation requires the use of type conversions.
   - It must have a method with the following header:

     ```
     public String toString()
     ```

     which returns a String of the form:

     ```
     CONDITIONS: info[0], Low: info[1], High: info[2], Range: info[3]
     ```

     where $info[i]$ is the value of the corresponding info field.

     Note: When an object is passed into a method such as `System.out.print()`, the String returned by the object's `toString()` method is printed in place of the object! All objects have the `toString()` method and you can override it to suit your needs, as you do in this part of the assignment.
   - All of its methods and data must be documented using Javadoc (see below for details).

7. Modify your Driver class main method so that it begins as follows:

```
public static void main(String[] args) throws IOException {
   BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
   String input = br.readLine();
```

8. Use the *input* string to create an instance of the Forecast class and print it out.
9. Test your code. Here are some examples. The inputs are to be typed into the console. Your program will then respond in the console with the output.

   Input:

   ```
   Sunny,29,53
   ```

   Output:

   ```
   CONDITIONS: SUNNY, Low: 29, High: 53, Range: 24
   ```

   Input:

   ```
   Rainy,38,42
   ```

   Output:

   ```
   CONDITIONS: RAINY, Low: 38, High: 42, Range: 4
   ```

10. Generate Javadoc using Eclipse.
    - Select *Project/Generate Javadoc...*
    - Make sure that your project is selected
    - Select *Private* visibility
    - Use the default destination folder
    - Click *Finish*
11. Open the *lab1/doc/index.html* file using your favorite web browser. Check to make sure that both of your classes are listed and that all of your documented methods have the necessary documentation.
12. If you complete the above instructions during lab, you may have your implementation checked by one of the TAs.

Submission instructions

- All required components (source code and compiled documentation) are due at 11:59pm on Friday, August 29.
- Prepare your submission file:
  1. Select the project in the *Package Explorer* window.
  2. Right-click. Select *Export*
  3. Select *General/Archive File*
  4. Expand your project and verify that both the *src* and *doc* folders are selected
  5. Enter the archive file name: *lab1.zip* (note that you may want to browse to a different destination folder)
  6. Select *Save in zip format*
  7. Click *Finish*
- Submit your zip file to the lab1 folder on D2L.

Hints

- See the **String** class API for how to split and interpret Strings.
- See the **Integer** class API for how to interpret Integers.

Project Documentation

At the top of every java source file for this course, you must include a documentation block at the top of the file that includes the following:

```
/**
   @author: <your name>
   Date: <>
   Project: <number>

   <A short, abstract description of the file>
*/
```

Note that the notation:

```
<some text>
```

indicates that you should provide some information and not actually write the less-than and greater than symbols.

Method-Level Documentation

Every method must include documentation above the method prototype using standard Javadoc markup. This documentation should be sufficient for another programmer to understand *what* the method does, but not *how* the method performs its task. For example, consider a method that will test whether a value is within a range and whose prototype is declared as follows:

```
public static boolean isInRange(double min, double max, double value)
```

The documentation for this method will be placed above the prototype in your java file and might look like this:

```
/**
   Indicate whether a value is within a range of values

   @param min Minimum value in the range
   @param max Maximum value in the range
   @param value The value being tested
   @return True if value is between min and max.  False if outside
```

```
this range.

*/
```

Note that this example includes all the required documentation elements:

- A short, intuitive description of what the method does (not how it does it),
- A list of the parameter names and a short description of the *meaning* of the parameter, and
- A short description of the return value.

Inline Documentation

Inside of each method, you must also include documentation that describes *how* a method is performing its task. This documentation should be detailed enough for another programmer to understand what you have done and to make modifications to your code. Typically, this documentation is preceded by "//" and occupies a line by itself ahead of the code that is being documented. While each line of code could be documented with its own documentation line, it is typically not necessary or appropriate. Instead, we typically use a single documentation line to capture what a small number of code lines is doing.

In addition, inline documentation should be done at a logical level and should not simply repeat what the line of code says.

Here is an example:

```
public static boolean isInRange(double min, double max, double value)
{
  // Check lower bound
  if(value < min)
     return false;

  // Check upper bound
  if(value > max)
     return false;

  // Within the boundaries
  return true;

}
```

Rubric

The project will be graded out of 100 points. The distribution is as follows:

Implementation: 35 points

Program formatting: 15 points

(15) The program is properly formatted (including indentation, curly brace and semicolon locations).

(8) There is one problem with program formatting.

(0) The program is not properly formatted.

Data types and method calls: 10 points

(10) The program is using proper data types and method calls.

(5) There is one error in data type or method call selection.

(0) There are multiple errors in data type and method call selection.

Required Methods: 10 points

(10) All of the required methods are implemented.

(0) The required methods are not implemented.

Proper Execution: 30 points

Output: 15 points

(15) The program passes all tests.

(10) The program fails one test.

(5) The program fails two tests.

(0) The program fails three or more tests.

Execution: 15 points

(15) The program executes with no errors.

(8) The program executes, but there is one minor error.

(0) The program does not execute.

Documentation and Submission: 35 points

Project Documentation: 5 points

(5) The java file contains all of the required documentation elements at the top of the file.

(3) The java file is missing one of the required documentation elements.

(2) The java file is missing two of the required documentation elements.

(0) The java file is missing more than two of the required documentation elements.

Method-Level Documentation: 10 points

(10) Every method contains all of the required documentation elements ahead of the method prototype.

(7) The method documentation is missing one of the required documentation elements.

(3) The method documentation is missing two of the required documentation elements.

(0) The method documentation is missing more than two of the required documentation elements.

Inline Documentation: 10 points

(10) Every method contains appropriate inline documentation.

(7) There is one missing or incorrect line of inline documentation.

(3) There are two missing or incorrect lines of inline documentation.

(0) There are more than two missing or incorrect lines of inline documentation.

Submission: 10 points

(10) The correct zip file name is used and has the correct contents.

(5) The correct zip file name is used, but one required component is missing.

(0) An incorrect zip file name is used or more than one required component is missing.