CS 2334, Spring 2017, Lab 1

Compilation, Debugging, and Javadoc Generation with Eclipse and Submission to Mimir

Due by: 4:00 pm CST, Friday, 27 January 2017

This lab is individual work. Each student must complete this assignment independently.

Objectives (Milestones):

1. Compile and debug a sample Java program and generate Javadoc documentation for it using the Eclipse IDE.
2. Submit the Java code to Mimir for grading, including automatic code testing.

Instructions:

Most students should already have a relatively current version of the Java Development Kit (JDK) and the Eclipse IDE installed. Therefore, most students should start with Step 4. If you do not have a recent JDK and Eclipse IDE installed, please start with Step 1.

1. Check to see if you already have a current version of the Java Development Kit (JDK) for Java 8 installed by opening a command or terminal window. (For Windows, click on *Start* | *Run* and then enter the text "`cmd`" and click "`OK`"; for Mac you can use the *Spotlight* to search for "Terminal" and click on the top hit. For Linux, you're probably already aware of how to start a terminal.) If you have the correct version installed and your path set correctly the following commands should give the results listed below the commands.

   ```
   >java -version
   java version "1.8.0_121"
   ...

   >javac -version
   javac javac 1.8.0_121
   ...
   ```

   If you already have a (relatively) current JDK8 installed, skip to Step 3. Otherwise, proceed to Step 2.

2. Install the JDK. For instructions for various operating systems, refer to the instruction files in the directory for this lab on the class website. Note that most of these are out of date—you should install Java 8 update 121.

   Once you have the JDK installed, check your installation to ensure everything is correctly installed and your `PATH` environment variable is correctly set by opening a command/terminal window and typing the commands as shown in Step 1. If everything checks out, proceed to Step 3. Otherwise seek help in completing Step 2.

3. Download and install Eclipse (https://eclipse.org/downloads/) using the installer and following the directions on the download page.

4. Download "`Lab1-eclipse.zip`" from the class website. Import this archive file into Eclipse using the instructions given in the "Basic Eclipse Tutorial" slides. This is in the first part of the slide set labeled "Lab2-slides." See, in particular, the slides marked "Add a .zip File As a New Project." (Note that this exercise was previously part of Lab 2, and the slide set has not been updated. Your Lab 2 will cover different material.)

5. Using Eclipse, add the following code to the `main` method of `Lab1Console.java` at the point indicated in the code:

   ```
   new Lab1Console(firstArg);
   ```

6. Eclipse should highlight some warnings and errors that you need to correct in the source code. Go through these and fix them all. Once you have removed all of these, the file should not contain any errors or warnings. (See, as necessary, the slides marked "Locate Errors in a .java File" from the Basic Eclipse Tutorial.)

7. Now, try to run the program by right-clicking on "`Lab1Console.java`" and select "Run as". Notice that the option to run as "Java Application" is not present. Determine the source of this problem and correct it in the program source code and verify that the program works by trying to run it again. Once you have successfully determined and corrected this problem, and you are convinced that the code can run, you are ready to run it.

8. Run `Lab1Console` as a Java Application. It should print the following message to the console:

   ```
   No argument provided. Enter the size for the array (an integer).
   ```

9. In response to this prompt, type in an integer value such as 10 and hit enter. `Lab1Console` will then produce a different type of error. Use the error information provided to you through Eclipse to track down and fix this error. Once this error is fixed, you should be able to provide an integer value at the console prompt and `Lab1Console` should run without errors and produce some simple (non-error) console output.

10. Once `Lab1Console` is running without errors using console input, you are ready to try *command-line arguments*, which are a standard way that one program running on a computer can talk to another program running on the same computer. This is very similar to the way that one method within a program can talk to another method—the calling method passes in arguments that are used for the parameters of the called method. This why the Java `main` method is set up as `public static void main(String[] args)`—it needs to be public so that it can be called from outside this class (and not just from another method in this same program but from another program altogether) and it needs to be able to take in a list of arguments (from that other program, if that program wants to talk to it).

    So, that explains what command-line arguments are but you may be wondering why they're called "command-line arguments" rather than something like "inter-program arguments" or "between-program arguments" or even just "program arguments." The reason is that, before people used graphical user interfaces (GUIs) to click on programs to make them run, they would, instead, use *command line interfaces* (CLIs), which are, as the name suggests, interfaces based on typing in lines of commands. So, to run a program, the user could type the name of the program into the CLI and hit enter. Alternately, if the user wanted to tell the program to behave in a special way when it runs, the user could type the name of the program followed by one or more arguments, then hit enter. Those other arguments were bundled up by the CLI and handed to the program to serve as that program's parameters to its `main` method. Because these "program arguments" were often seen by users as a way to get their command-line instructions to the programs they were running, they referred to them as "command-line arguments."

    For this lab, you don't need to use a CLI to try out command line arguments (although you could—see Step 1 for how to open a CLI on your computer). Instead, Eclipse provides a way for you to put in "command-line arguments" (which it quite properly calls "program arguments"). See the slides in the Basic Ecplise Tutorial marked "Run a Project with Command Line Arguments" for instructions.

    Try the following several argument sets, each in turn, and keep track of what happens each time. When you submit your completed lab to Mimir (see below), you'll also take a quiz that will ask you how `Lab1Console` responded for each of these argument sets.
    a. `alpha`
    b. `alpha 1`
    c. `1 alpha`
    d. `1.5`
    e. `10`

11. Once you have tried out command-line arguments, it's time to generate *Javadoc*, which is html documentation of Java code that is built from properly formatted comments within the Java source code itself. This html

documentation can then be put online and easily browsed with a standard web browser or viewed from within an IDE like Eclipse. Done properly, this makes it easy for software developers using this Java code to know what they need to know about it without having to have access to or to read through the code itself.

The various comments starting with a slash and two asterisks (`/**`), rather than a slash and just one asterisk like normal multiline comments, are processed by the Javadoc generator to create a separate html documentation page for each class. This has the nice feature that the html documentation for a class can be easily kept in sync with the comments found in the code itself. And, since the developer should keep the comments in the code up to date as the code itself changes, this means that it should be easy to keep the html documentation for the code up to date as well. Of course, if the developer *doesn't* keep the comments in sync with the code or *doesn't* run the Javadoc generator over the code when changes are made, it would still be possible for the html documentation to get out of sync with the code itself.

Follow the instructions in the Basic Eclipse Tutorial marked "Create Javadoc" to (attempt to) create Javadoc for `Lab1Console.java`.

12. When you run the Javadoc generator, it should flag an error. This is because there is an error in one of the comments—it doesn't match the code itself. That's right, the Javadoc generator is actually smart enough to check the comments to see if they match the code and refuse to generate bogus Javadoc! Okay, the Javadoc generator isn't smart enough to catch every possible error. It won't, for example, try to parse long English sentences to see if they accurately describe what the code does, but it will look for simple things, like whether the parameters marked `@param` in the comments match up with actual parameters in the code.

13. Find the discrepancy between the comments and the code, and fix it. Then rerun the Javadoc generator to create the Javadoc again. If you fixed the error, it should run fine and generate a nice set of Javadoc files for `Lab1Console`.

14. Open up the `doc` directory that the Javadoc generator created inside your `Lab1ConsoleProject` directory, find the `Lab1Console.html` file, and open that in Eclipse. Scroll through it and try to understand how the Javadoc matches the comments from the code. You will submit this file to Mimir later.

15. Once your Javadoc is successfully created, it's time to try out the Eclipse debugger. See the Basic Eclipse Debugging slides in the Lab2-slides file for instructions on how to add breakpoints and run the debugger. *Breakpoints* are places in the code that it should temporarily pause execution so that you can see what is happening inside the program. The *debugger* is a tool that allows you to see what is happening inside a program by showing the values of the variables as the code executes. You use them together by marking breakpoints at places in the code that you want to examine and then running the debugger which will execute the code up to the first breakpoint. You can then look at the variables you want to know about and either jump to the next breakpoint or step forward line by line, examining the changes to the variables as you go.

    One variable you should care about is `myArray`, in particular, the fifth entry in the variable `myArray`, that is, `myArray[4]`. This is because you will be quizzed through Mimir about the value of this variable at various places in the code. Set appropriate breakpoints and use the debugger to find the value for this variable at the following places in the code.
    a. After `myArray` is created but before the call to `init()`.
    b. After the call to `init()` but before the call to `display()`.
    c. After the call to `display()` but before the call to `work()`.
    d. In the call to `work()` when the value of `index` is `4`.
    e. After the call to `work()` but before the call to `System.out.println()`.

16. Finally, it is time to submit to Mimir for automatic grading. Go to https://class.mimir.io, create an account

using your OU email address, and log in. Then use the course code "`a5fce6c76e`" to add yourself to the appropriate course.

17. Once you are logged in to the correct course, choose the project "CS 2334 Spring 2017 Lab 1 Console" and click "Submit Files." From the file selection menu, navigate to Lab1Console.java inside your Eclipse workspace and submit that file. If you have done everything correctly, Mimir will run three tests on your code —one for I/O, one for use of command-line arguments, and one for code quality—and your code will pass all three tests.

18. If your code does not pass all three tests, click on the gear icon next to each test that wasn't passed to see what your mistakes were. Return to Eclipse to fix these. When you are convinced that you have corrected all mistakes, go back to the previous step and submit again. Continue in this loop until your code has passed all tests or until you have exceeded the five tries limit for this lab. Then go on to the next step.

19. After your code passes all the test, it's time to test your Javadoc. Select "CS 2334 Spring 2017 Lab 1 Javadoc" and submit your `Lab1Console.html` file. Mimir should evaluate your file to see if it matches the one generated by Prof Hougen. If it does, your Javadoc passes the test. If not, try to determine what you might have done wrong, fix it, and resubmit. You'll have five tries for this as well.

20. After your code and Javadoc have passed their tests, it's time to see if your use of command-line arguments (program arguments) and the debugger gave you the correct information. Select "CS 2334 Spring 2017 Lab 1 Quiz" and take it. Once you're done with the quiz, you're done with this lab!