LAB 3

Learning Outcomes

- Hand-trace a program to anticipate the flow of execution
- Analyze the effects of try-catch statements on the overall program execution
- Locate the source of errors using the messages displayed in the console
- Use the Debugger Tool in Eclipse to step through code and track variable values to understand the reason for such errors
- Fix the errors that have been found in the code to make it run as expected

Pre-Lab

- Create a new Java project called Lab3
- Download the files: Exceptions.java, DebuggingExercise1.java, DebuggingExercise2.java, DebuggingExercise3.java, and MyObject.java
- Save these downloaded files into the Lab3 src folder

Exercise 1 – Follow the exception handling execution

1. Open the Exceptions.java file in Eclipse. Do not run it yet.
2. Examine the code to think about the flow of execution.
3. Jot down what you think the program would output to the console from each of the 3 cases (the calls to method1() from the main method).
4. Once you have finished hand-tracing the program, run it to see the actual output.
5. Were you correct? If not, read through the code again to understand why it executed this way.
6. Now comment out Line 10 (`P[x] = 5`) and un-comment Line 11 (`method2(P, x)`).
7. Read the code within method2() and hand-trace the execution on paper with this small change in the code.
8. Once you have thought about the expected output, run the program to see the actual output.
9. Were you correct? If not, read through the code again to understand why it executed this way.

Exercise 2 – Using the Debugging Tool

1. Open DebuggingExercise1.java and run it. You will see that it compiles but it crashes when you run it. Read the error message and try to understand it. It will give you some information that may help you find the source of the error.
2. Check that you have a Debug button in the top right corner of your Eclipse window (a button with a little green bug in it), beside the Java button. If you don't, select Window > Perspective > Open Perspective > Debug.
3. Click on the Debug button to change to the Debug view. You can, at any time, switch back to the Java view by clicking on the Java button in the top right corner (the button with the blue letter J located to the left of the debug button).
4. In the window displaying your code, add a breakpoint to the line `for (int j=1; j<=6; j++)`. You do this by right-clicking in this line at the very left, and then clicking Toggle Breakpoint. You can remove a breakpoint at any time in the exact same way.
5. When executing your program, the debugger will stop every time this line is reached. Run your program in the debug mode by selecting on Run > Debug (alternatively, press the key F11).
6. You will see that the line `for (int j=1; j<=6; j++)` is highlighted in green. The execution of the program has stopped at this point. In the Variables window (upper right corner) you will see the variables of the program: args, testArray, and i and their values. Since testArray is a variable referencing a 2-dimensional array, or an array in which every entry is an array of integers, when you click on the ">" symbol to its left, the debugger shows 5 new variables ([0], [1], [2], [3], and [4]), each one of them references an array of length 6; these are the rows of the 2-dimensional array. If you click on the ">" symbol to the left of [0], six new variables will be shown; these are the columns of the first row of the two dimensional matrix.
7. Now you can use F5 to execute your program one instruction at a time (this is called single-stepping). Press F5 once. Note how the variables `i` and `j`, and the values stored in the 2-dimensional array are changing in the Variables window. Why did testArray[0][0] not change?
8. Keep pressing F5 and stop when the value of testArray[0][5] changes value from 0 to 5 and the statement `for (int j=1; j<=6; j++)` is highlighted in green. Push key F5 one more time. What are the values of i and j? Is it correct that the program tries to store the value `(i+1)*j` in `testArray[i][j]`?
9. After answering the previous questions, you should know what is wrong with the program. Fix the error. Run the program and make it sure that it does not crash; it should print 30 values.
10. Modify the code so all entries of testArray are properly initialized. The first row of testArray should store the values: 1, 2, 3, 4, 5, 6; the second row should store the values: 2, 4, 6, 8, 10, 12; the third row: 3, 6, 9, 12, 15, 18; the fourth row: 4, 8, 12, 16, 20, 24, and the last row: 5, 10, 15, 20, 25, 30.

Exercise 3 – Using the Debugging Tool

1. For this and the following exercises you CANNOT add any print or println statements to the code that you are testing.
2. Open DebuggingExercise2.java and run it. It will also crash.
3. Add a breakpoint to the line `i = process(2);`
4. Run the program in Debug mode.
5. Keep pressing F5 until you reach the statement `return result;` (do not execute this statement yet).
6. Using the debugger, find out the values of the variables: i, step, result, and load.
7. Since load is a static variable, to see its value in the Variables window you need to click on the little inverted triangle in the upper right corner of the Variables window; this will bring two options: "Layout" and "Java". Click on Java and then select "Show static variables" (see a screenshot of this option).
8. Terminate the program by selecting Run > Terminate.
9. Run this program again and stop execution at statement `i = process(2)`.
10. This time press the key F6 instead of the key F5. You will see that the debugger now does not go inside method `process()`, but it goes directly to statement `total = (i * 100) / load` (key F5 tells the debugger to "step into" a method, while key F6 makes the debugger "step over" a method). What is the value of load? Why does the program crash if you keep pressing F6? (Do not press F5 or the debugger will try to step into the code of the Java's virtual machine).
11. Fix the program so that it does not crash and it prints the message "The value of total is infinity". Change the statement `i = process(2)` to `i = process(1)` and run the program again; this time it must print "The value of total is 5". (hint: check if load is 0 before computing `total = (i * 100) / load`).

Exercise 4 – Using the Debugging Tool

1. Open DebuggingExercise3.java and run it. It will not compile.
2. Which line causes the compilation error? Why does the compiler complain about this line even though i has been declared in the statement: `for (int i = 1; i < 2; ++i)` (hint: what is the scope of i?)
3. Fix the code by commenting out the line with the compilation error.
4. Run the program; it should not produce any output.
5. Study the code for this class and try to determine without running the program what the values for the variables var1 and obj1 are immediately after `method2(obj1)` is executed. Write down your answers. Now try to determine the values of the variables immediately after `method1(var1)` is executed. Write down your answers again.

6. Now use the debugger to determine the actual values of the above variables. To see the string stored in obj1, in the Variables window click on the ">" symbol to the left of obj1; you will see now the value of name.
7. Were you correct? If not, study the code and try to understand why the variables have the values shown by the debugger.
8. Add a breakpoint at the line `if (obj1 == obj2)`. Press F11 to run the program in debugging mode and stop at this statement.
9. Press F5. Why is the result of the comparison false even though both obj1 and obj2 contain the same information, namely "joe"?
10. Look at the value of obj1 in the Variables window of the debugger. Since obj1 is a non-primitive variable its value is the address of an object of the class MyObject. Note that the debugger does not show the actual address of the object referenced by obj1; instead it displays the address in a user-friendly format: MyObject (id = 19) —you might get a different value for id. This means that the address stored in obj1 is the address of an object of class MyObject and the debugger assigns it a unique internal identifier (in the above example the identifier is 19). If you were to print the value of obj1 with `System.out.println(obj1)` you would get a similar output, not the actual address of the object but a string of the form MyObject@5ccd43c2; the Java virtual machine will assign a different identifier to the object (5ccd43c2) than the debugger.

Submission

When you have completed the lab, navigate to the weekly module page on OWL and click the Lab link (where you found this document). Make sure you are in the page for the correct lab. Upload the files listed below and remember to hit Save and Submit. Check that your submission went through and look for an automatic OWL email to verify that it was submitted successfully.

Rules

- Please only submit the files specified below. Do not attach other files even if they were part of the lab.
- Do not ZIP or use any other form of compressed file for your files. Attach them individually.
- Submit the lab on time. Late submissions will receive a penalty.
- Forgetting to hit "Submit" is not a valid excuse for submitting late.
- Submitting the files in an incorrect submission page will receive a penalty.
- You may re-submit code if your previous submission was not complete or correct, however, re-submissions after the regular lab deadline will receive a penalty.

LAB 3

Files to submit

- DebuggingExercise1.java
- DebuggingExercise2.java
- DebuggingExercise3.java