Lab 7: Debugging and Unit Test!

# Let’s get started!

Today, we’d like you to:

1. **Open Eclipse**. Remember to keep all your projects in the same workspace. Think of a workspace as a shelf, and each project is a binder of stuff on the shelf. If you need help organising your workspace, we’d be happy to help!
2. **Create a new Non-Modular Java project**. Call it Comp1510Lab07*LastNameFirstInitial*
3. **Create** a Microsoft Word document Comp1510Lab07*LastNameFirstInitial.*docx to hold your answers to the questions below (wherever the lab below wants a screenshot or answer to a question).
4. **Complete the following tasks**. Remember to right-click the src file in your lab 7 project to quickly create a new package and Java class.
5. When you have completed the exercises, **show them to your lab instructor**. Be prepared to answer some questions about your code and your choices.

# What will you DO in this lab?

In this lab, you will:

1. Familiarize yourself with the Eclipse debugging interface and common debugging tasks, such as:
   1. Setting breakpoints and conditions
   2. Looking at variables and the call stack
   3. Setting variables
2. Practice debug skills with examples
3. Write Junit test code

[LET’S GET STARTED! 1](#_bookmark0)

[WHAT WILL YOU DO IN THIS LAB? 1](#_bookmark1)

1. [GETTING FAMILIAR WITH DEBUGGING IN ECLIPSE 2](#_bookmark2)
2. [THE DEBUG CHALLENGE 9](#_bookmark3)
3. [WRITING UNIT TESTS WITH JUNIT. 10](#_bookmark4)
4. [YOU’RE DONE! SHOW YOUR LAB INSTRUCTOR YOUR WORK. 12](#_bookmark5)

# Getting Familiar with Debugging in Eclipse

In this exercise, you will use the Eclipse IDE (Integrated Development Environment) for Java development. You are already familiar with Eclipse. Once you understand one IDE, you should also be able to work your way around others.

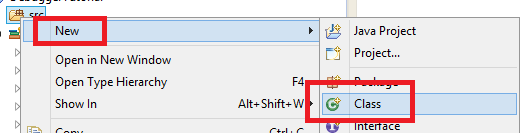
### Look at your new project.

Eclipse will create the following structure for you.



By default, no source code files are created for you. The IDE also gives you access to a variety of libraries represented by .jar package files.

### Create Package and Class file

* + 1. Create a Class File template.

We need to add a class named DebugStar in the package ca.bcit.comp1510.lab07 with a main() method. Right-click on src and choose

New >> Class. Package: ca.bcit.comp1510.lab07 Name: DebugStar

Check the box to create a main method header. The following class is created for you:

package ca.bcit.cst; public class DebugStar {

public static void main(String[] args) {

// TODO Auto-generated method stub

}

}

* + 1. Create Class file content. Replace the entire main() method with the following code:

public static void main(String[] args) { run("+", 6, 7);

run("-", 6, 7);

run(6);

}

private static Operation getOperation(final String key) { final Operation operation;

if(key.equals("+")) {

operation = new Add();

} else {

operation = new Subtract();

}

return (operation);

}

private static void run(final String key, final int a, final int b) { final Operation operation;

final int result;

operation = getOperation(key); result = operation.perform(a, b);

System.out.println("result = " + result);

}

private static void run(final int n) { final Factorial factorial; final int result;

factorial = new Factorial(); result = factorial.perform(n); System.out.println("result = " + result);

}

It has one main method and three private static methods.

* + 1. Add additional Classes and an Interface Definition. Add the following code at the very bottom end of the file, **outside of the class definition**.

interface Operation {

int perform(int a, int b);

}

class Add implements Operation { @Override

public int perform(final int a, final int b)

{

return (a + b);

}

}

class Subtract implements Operation { @Override

public int perform(final int a, final int b) { return (a - b);

}

}

class Factorial {

int perform(final int n) {

int ret; ret = 1;

for(int i = 1; i < n; i++) {

ret \*= i;

}

return (ret);

}

}

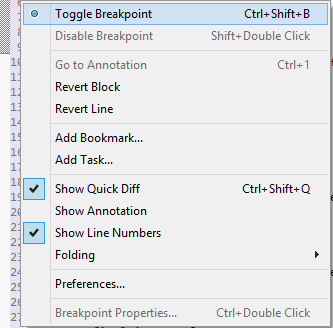
### Run Program.

Run the program and look at the console output. The above program essentially does 6 + 7, then 6 – 7, then 6! (6 factorial). This program is convoluted to highlight some debugger features. We can use the debugger to help us understand what the program is doing.

### Prepare for Debugging: Set Breakpoint

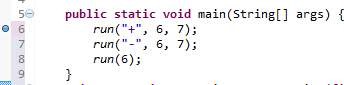
First, make sure that line numbers are displayed in your source files. In Eclipse, go to

WindowPreferencesGeneralText Editor check box for line number.



The most essential thing you will do in a debugger is use a ***breakpoint***. A *breakpoint* is a line (point) in your program that you want the debugger to pause (break) at. There are a few ways to set a breakpoint; one of them is to right-click at the line you want to set the breakpoint on (in this case, the very first line in the main method) and select the Toggle Breakpoint from the sub-menu.

Set a breakpoint on the first run method call. You will now see a blue dot on the left-hand side of the line at which the program will pause when it is executed.

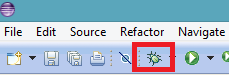


You can disable the breakpoint by right-clicking on the dot and toggling the breakpoint. You will see that the breakpoint disappears. Click the same spot again and you will see that the breakpoint comes back. You can also toggle a breakpoint by hitting **Ctrl-Shift-B** on the keyboard (or CMD-Shift-B on SX).

### Start Debugger and View Variables

Now, launch the debug mode.

There are several possible ways to do this.

* Menu Run  Debug
* Click on the “insect” icon on the toolbar
* Select the source file, Right Click Debug As 

Java Application

* Or Hit F11 on your keyboard

|  |  |
| --- | --- |
|  | Click on “Yes” if you see this dialogue because we certainly wish to see the Debug perspective.  Now, run your program in debug mode. Execution will stop when it hits the breakpoint, which is the first line of the program in this case. |
|  | When a breakpoint is hit, you will see a green line. The program is now paused at that line, waiting for you to do something. |

|  |  |
| --- | --- |
|  | The top right-side view in Eclipse is the *Variables* window that shows you the value of all the variables that the current method has access to (scope).  If you do not see the variables window, then use the *Window * *Show View* menu to access the different windows that are available. |
|  | If you wish to view the type of the variable, click on the down arrow at the top of the Variables View window. Then, choose “Select Columns” and click on the “Declared Type” checkbox. |

### Debugging: Step Through Code

The most common activity you will probably use in the debugger is to step through the code. There are several options, such as

* stepping over a line (F6)



Step Over

Resume Terminate

Step Into

* stepping into a method (F5)
* resume until the next breakpoint is hit (F8)
* terminate (CTRL + F2)

Select “Step Into” or just press F5.

As you step into methods, the *Variables* window updates the variables that the current method has access to. Here, you can see that the variables have values for keys a and b.



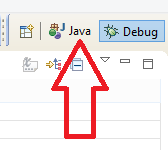
You can even change the values if you want to (don’t do that for this tutorial, though).

“Step Over (F6)” is used to execute individual lines of code. If you “Step Over” a method

call, then the method is run, and the debugger continues on after the method call.

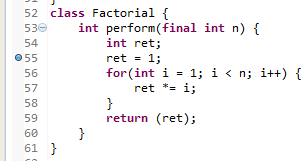
As you step through the code the green line updates as does any variable values. This is a great way to see how your code works.

Just keep stepping through the code and get comfortable with that. Keep going until the program ends.

To return to your source code, click on the “Java” tab in the top right-hand corner of your screen.

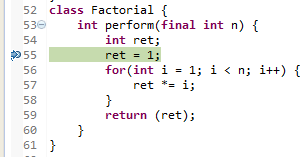
### Debugging: Looking at Call Stack, Using “Step-Return”

* For the next part of the tutorial, turn off all breakpoints that you have set.



* Now set a new one on the “ret = 1;” line in the factorial class perform() method.

Now, run the program in the debugger. The program will run and stop at the breakpoint. This is useful when you want to start debugging at a certain point in the program. Carefully step through the “for” loop.

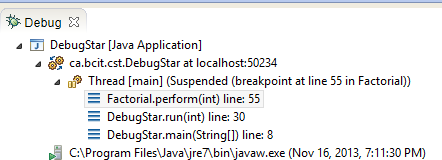


Each time you step, look at the variables. In the editor window, you can also select a variable name to show the value.

Notice that the Variables update as you step through the “for loop”. Make sure you stop

stepping on the return statement.

Have a look at the *Call Stack* window on the top left-hand side that shows how you arrived at the current line (the series of method calls that got you to where you are).

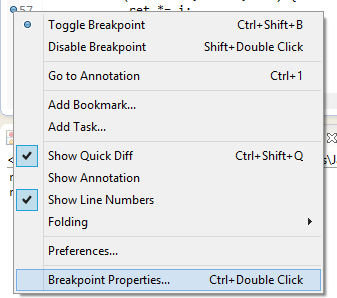


Use the Call Stack window to see what method calls were made. When you select a line on the Call Stack window, you will be taken to the method call.

Terminate and repeat the process to debug again. However, this time, use “Step-Return” before the “for” loop is over. What happens?

### Debugging: Breakpoint on Condition

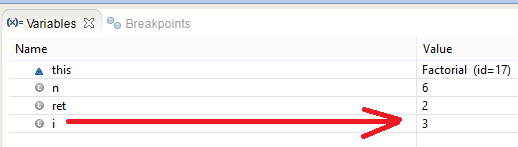
Stop the debugger and delete the existing breakpoint. Set a new breakpoint **inside** of the factorial for-loop.



Right-click on the breakpoint you set and choose “Breakpoint Properties…”

You can set a condition for when the debugger will break. There are several ways to set a condition. You can set the “Hit Count” to be 3. After your matched setting is shown, click on the OK button.

What happens when you run the program in the debugger now? The program will stop on the breakpoint when “i” hits the value of 3. It is useful when you only want to break when a certain value is reached.



Now, set a “conditional” breakpoint so that it stops at the condition i==5 instead. Run the program until it stops at the conditional breakpoint. Show a screen capture of the Variables Windows from the Debug Perspective.

## SCREEN CAPTURE ONE

**SHOW VARIABLES WINDOW**

文本

描述已自动生成

### Time to test your understanding! 

## TABLE ONE

|  |
| --- |
| Question: What is the Eclipse keyboard shortcut for toggling a breakpoint? |
| Answer: **Ctrl+Shift+B** |
| Question: What is the difference between “Step-Over” and “Step-Into” and “Step-Return”? |
| Answer:  Step-Over: Execute the complete function.  Step-Into: Enter the function and have the user operation execute the function line by line.  Step-Return: Execute all the remaining lines and pause at the return point. |
| Task: Practice tracing through the DebugStar sample program.  It is ok if you don’t understand all of the Java code, but you should be able to trace the order in which statements are executed.  Based on your best understanding of the program, provide a list of methods that are called when the program executes (from start to end, in order of being called). You can skip library methods (like println, for example).  HINT: Use a combination of “Step-Into”, “Step-Over”, and “Step-Return”. Use the “Stack Trace” window. |
| List of Methods (in order of call) below. Please use the fully qualified name, eg.  “DebugStar.run(String, int, int). Use the stack view to help you.  Debugstar.main(String[] args);  //run(“+”, 6, 7);  Debugstar.run(final String key, final int a, final int b);  Debugstar.getOperation(final String key);  Add(); //Constructor  Add.perform(final int a, final int b);  //run(“-“, 6, 7);  Debugstar.run(final String key, final int a, final int b);  Debugstar.getOpetation(final String key);  Subtrack();  Add.perform(final int a, final int b);  //run(6);  Factorial();  Factorial.perform(final int n); |

# The Debug Challenge

### You are given a source code for a program that is “BUGGY” called FibonacciBuggy. Your job is to find out why, using the skills that you have acquired so far.

* 1. Provide a screen capture of the original code (with line numbers)

## SCREEN CAPTURE: Original Code with line numbers

Screen Capture of Original Code

文本

描述已自动生成

* 1. Provide an error log table (such as the one below) indicating error details (line number, type of error, and explain the error and show correction).

## TABLE TWO

|  |  |  |  |
| --- | --- | --- | --- |
| **Line Number** | **Type of error (compile-time, run-time, or logical)** | **Description** | **Correction** |
| **Some places may have incorrect capitalisation; please ignore them. Microsoft documents always "help" me by automatically changing lowercase to uppercase. I've corrected many, but there might still be a few that I overlooked.** | | | |
| 26 | Compile-time | Need “;” | int n; |
| 27 | Compile-time | Variable “n” may not have been initialised | do{//code block} while (condition); |
| 27 | Logical | Expect input > 2, so loop condition should be not > 2, which is <= 2 | While (n <= 2) |
| 29 | Run-time | “nexInt” method will throw “*InputMismatchException* *(nfe.getMessage())” when catching “NumberFormatException nfe”, and the program will shut down if there is no method to solve this throw.* | while(true) {  try {//code block  break;  } catch (Exception ignored) {  System.out.println(“please enter a number: ”);  scanner.nextLine();  } |
| 35 | optimisation | The explicit type argument Integer can be replaced with <> (diamond operator) | List<Integer> f = new ArrayList<>(n); |
| 40 | Logical | The first number should be 0 but not 1, so if the first get method f.get(i - 1) wants to get the first number of f, it needs f.get(0); in the first loop, i == 2, so i – x = 0, x = 2 | f.add(f.get(i – 2) + f.get(i – 1)); |
| 47-51 | Logical | Idk why the initialised number is 2. If you want to get the first number of the list, you need the parameters of the “get” method in line 49, which is “0” in the first loop, which means the “i” in the parameter was supposed to be 0. However, the condition in 48 is “i <= size of the list”. Because list positions start from 0, the list size is always one less than the list position. Therefore, “i” in the condition was supposed to be 1. | Solution1:  int i = 0;  while (i <= fiboList.size() - 1) {  System.out.print(fiboList.get(i));  i++;  }  Solution2:  int i = 1;  while (i <= fiboList.size()) {  System.out.print(fiboList.get(i – 1));  i++;  }  Solution3:  int i = 0;  while (i + 1 <= fiboList.size()) {  System.out.print(fiboList.get(i));  i++;  } |
| 49 | Logical | Commas and spaces need to be inserted to separate the numbers. | System.out.print(fiboList.get(i) + ", "); |

* 1. Provide screen capture of fixed code (with line numbers) and sample run, using n=10:

## SCREEN CAPTURE: Fixed code with line numbers

**Screen Capture of Fixed Code and Console Output**

文本

描述已自动生成

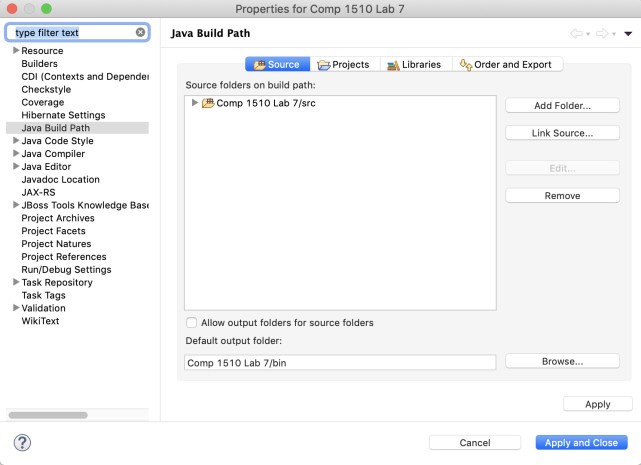


# Writing Unit tests with JUnit.

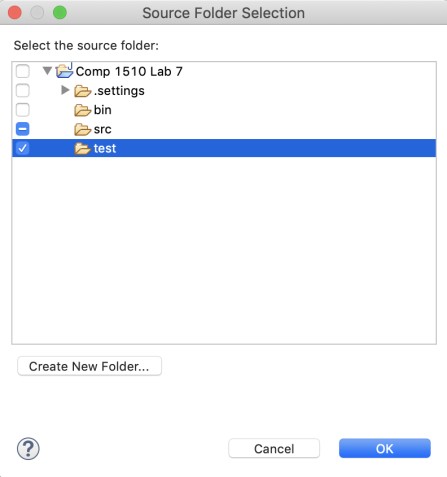
After all that debugging, we may not have a lot of time to write unit tests, so this section will be shorter.

* 1. Add a test folder to hold your tests. This is not strictly needed but keeps the test code separated from the code being tested. Right-click on your project and select Build Path

> Configure Build Path … Then click on the src tab. You should see the following:



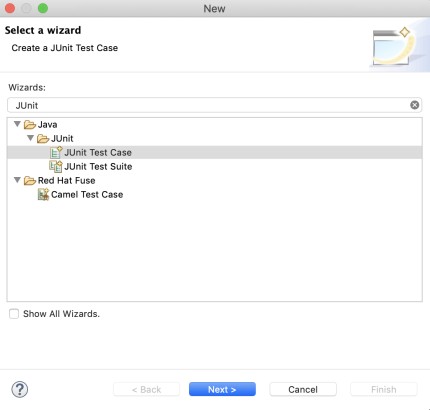
* 1. Now click on Add Folder, then click on Create New Folder and enter *test* in the Folder name field that pops up. The result should look as follows:



* 1. Now click on the **OK** button, then the **Apply and Close** button on the Build Path

Window. You should see a test folder in your project that looks like The symbol on the folder indicates you have set up a new source folder (if no folders have this symbol, then Eclipse will not compile any of your code, resulting in massive confusion).

* 1. From now on, you should put the unit test code under a test folder and your regular code under the src folder. Add a package ca.bcit.comp1510.lab07 to the test folder.
  2. From the learning hub, download the file TestThis.java and copy it into your src folder in the ca.bcit.comp1510.lab07 package. This will be what we are testing. There are two methods, both finding the largest value of integers.
  3. Right-click on the package in the test folder, choose New > other, and type JUnit in the text field as follows:



Then select Junit Test Case, and in the New Junit Test Case wizard, select New Junit Jupiter Test, enter the test class name (TestThisTest) and the class under test (TestThis) and click Next.

* 1. On the next screen, under TestThis, click the checkboxes in front of the two methods (largest() with different parameters) and click Finish.
  2. If you have not already added Junit to your build path, click OK on the message that pops up (or add Junit 5 to the library manually).
  3. Now, it is a matter of adding test methods. Each method should be called one of the methods of the TestThis class, and an assert method (such as assertEquals) should be used to check the correct result. Each method should have a name indicating what is being tested. You can cut and paste the ones provided by Eclipse and add numbers at the end.
  4. Start with methods named, for example (you can come up with better names):
     1. testLargestIntIntInt1
     2. testLargestIntIntInt2
     3. testLargestIntIntInt3
     4. testLargestListOfInteger1
     5. testLargestListOfInteger2
     6. testLargestListOfInteger1
     7. and so on.
  5. Fix any errors your tests find.
  6. When you think you have done enough testing, run the Coverage tool to see if you need more. You should be able to get complete coverage and all tests green!

# You’re done! Submit your work.

If your instructor wants you to submit your work in the learning hub, export it into a Zip file in the following manner:

* 1. Right-click the project in the Package Explorer window and select export…
  2. In the export window that opens, under the General Folder, select Archive File and click Next
  3. In the next window, your project should be selected. If not, click it.
  4. Click *Browse* after the “to archive file” box, enter the name of a zip file (the same as your project name above with a zip extension, such as Comp1510Lab07BloggsF.zip if your name is Fred Bloggs) and select a folder to save it. Save should take you back to the archive file wizard with the full path to fill in the saved file. Then click Finish to actually save it.
  5. Submit the resulting export file *and* your answers document as the instructor tells you.