

# 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 some help organizing your workspace, we'd be happy to help!
2. **Create a new Java project.** Call it something that will help keep your code organized, i.e., COMP 1510 Lab 7.
3. **Complete the following tasks.** Remember you can right-click the src file in your lab 7 project to quickly create a new package and Java class.
4. When you have completed the exercises, **show them to your lab instructor.** Be prepared to answer some questions about your code and the choices you made.

## 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:
  - a. Setting breakpoints and conditions
  - b. Looking at variables and the call stack
  - c. Setting variablea
2. Practice debug skills with examples
3. Write Junit test code

## Table of Contents

<b>Let's get started!</b>	<b>1</b>
<b>What will you DO in this lab?</b>	<b>1</b>
1. <b>Mathematics</b>	Error! Bookmark not defined.
2. <b>Enhancing the Name class</b>	Error! Bookmark not defined.
3. <b>You're done! Show your lab instructor your work.</b>	<b>10</b>

## 1. Getting Familiar with Debugging in Eclipse

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

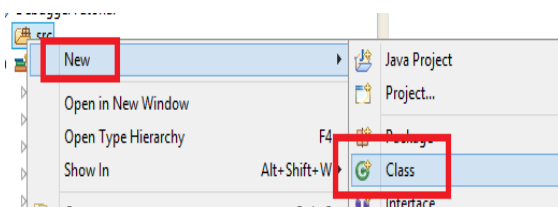
### 1. Look at your new project.

Eclipse will create the following structure for you.



No source code files are created for you by default. Also, the IDE gives to access to a variety of libraries that are represented by **.jar** package files.

### 2. Create Package and Class file



**DebugStar**

#### a. Create Class File template.

We need to add a class named **DebugStar** in package **ca.bcit.comp1510.lab7** that has a **main()** method. Right-click on **src** and choose **New >> Class**. Package: **ca.bcit.comp1510.lab7** Name:

Check the box for creating 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
    }
}
```

- b. 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 3 private static methods.

- c. 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);
    }
}

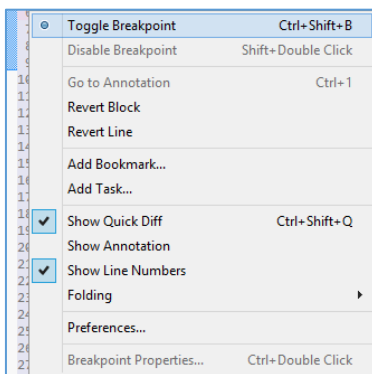
```

### 3. 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 allow us to highlight some debugger features. We can use the debugger to help us understand what the program is doing.

### 4. Prepare for Debugging: Set Breakpoint

First, make sure that line numbers are displayed in your sources files. In Eclipse, go to Window → Preferences → General → Text Editor → check box for line number.



The most basic 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.

```

5 public static void main(String[] args) {
6     run("+", 6, 7);
7     run("-", 6, 7);
8     run(6);
9 }

```

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.

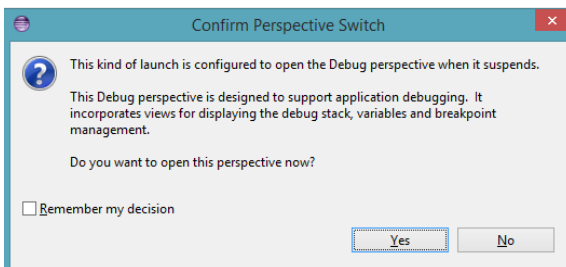
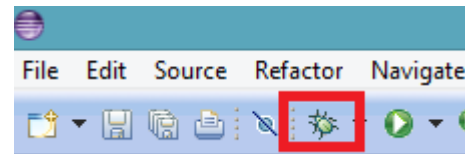
You can disable the break point by right-clicking on the dot and toggling the breakpoint. You will see that the breakpoint disappears. Click the same spot again and you will see that the breakpoint comes back. You can also toggle a breakpoint by hitting **Ctrl-Shift-B** on the keyboard (or **CMD-Shift-B**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

## 5. 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 source file, Right Click → Debug As → Java Application
- ❖ Or Hit F11 on your keyboard



Click on “Yes” if you see this dialog because we certainly wish to see the Debug perspective.

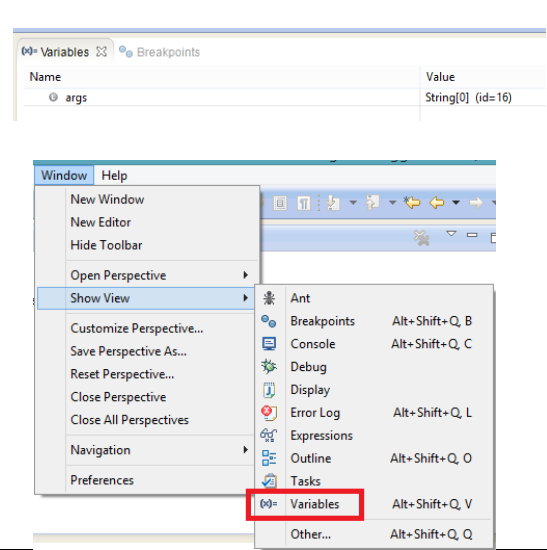
Now, run your program in debug mode. Execution will stop when it hits the breakpoint, which is the first line of the program in this case.

```

DebugStar.java
1 package ca.bcit.cst;
2
3 public class DebugStar {
4
5     public static void main(String[] args) {
6         run("+", 6, 7);
7         run("-", 6, 7);
8         run(6);
9     }
10
11     private static Operation getOperation(final String key) {
12         final Operation operation;
13         if(key.equals("+")) {
14             operation = new Add();
15         } else {
16             operation = new Subtract();
17         }
18         return (operation);
19     }
20 }

```

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.

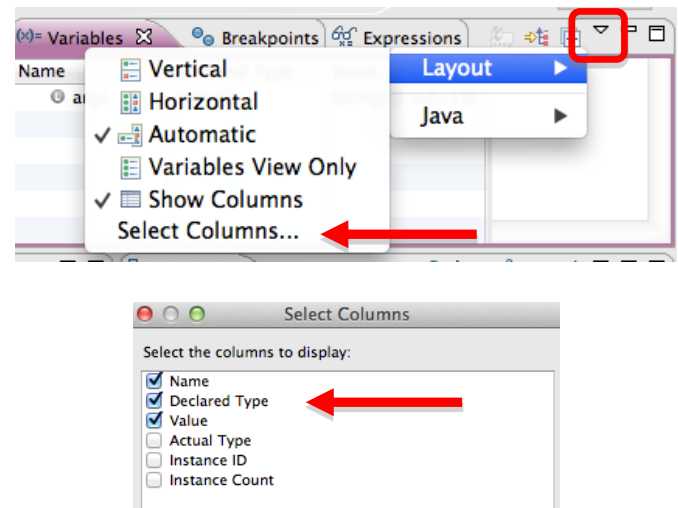


The top right-side view in Eclipse is the *Variables* window that shows you the value of all the variables that the current method has access to (scope).

If you do not see the variables window, then use the *Window* → *Show View* menu to access the different windows that are available.

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.



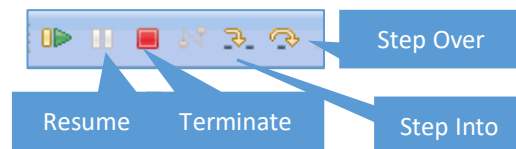
If you wish to view the type of the variable, click on the down arrow at the top of the Variables View window. Then chose “Select Columns”, and click on the “Declared Type” checkbox.

If you wish to view the type of the variable, click on the down arrow at the top of the Variables View window. Then chose “Select Columns”, and click on the “Declared Type” checkbox.

## 6. 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)
- stepping into a method (F5)
- resume until the next breakpoint is hit (F8)
- terminate (CTRL + F2)



Select “**Step Into**” or just press F5.

Variables	
Name	Value
key	"+" (id=403)
a	6
b	7

As you step into methods, the *Variables* window updates the variables that the current method has access to. Here you can

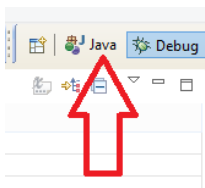
see that the variables have values for key, a, and b.

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.

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

```

52 class Factorial {
53     int perform(final int n) {
54         int ret;
55         ret = 1;
56         for(int i = 1; i < n; i++) {
57             ret *= i;
58         }
59         return (ret);
60     }
61 }

```

- 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.

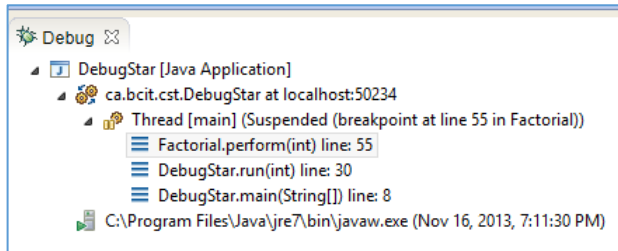
```

52 class Factorial {
53     int perform(final int n) {
54         int ret;
55         ret = 1;
56         for(int i = 1; i < n; i++) {
57             ret *= i;
58         }
59         return (ret);
60     }
61 }

```

Now run the program in the debugger. The program will run and stop on the breakpoint. This is useful when you want to start debugging at a certain point of 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.

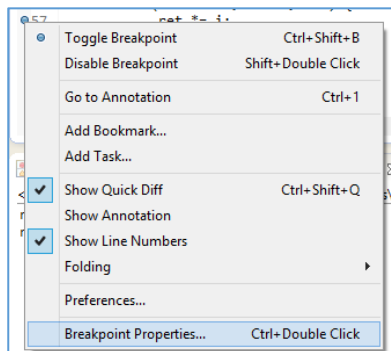


Have a look at the *Call Stack* window in the top left-hand side that shows how you arrived at the current line (the series of method calls that got you to where you are).

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?

## 8. Debugging: Breakpoint on Condition

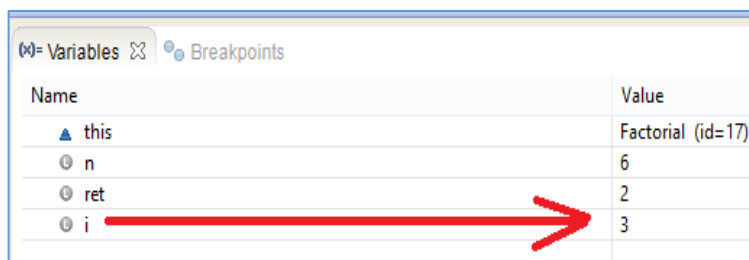


Stop the debugger and delete the existing breakpoint.

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 as 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  
HERE BELOW:**



**9. Time to test your understanding! ☺****TABLE ONE**

Question: What is the Eclipse keyboard shortcut for toggling a breakpoint?
Answer:
Question: What is the difference between “Step-Over”, and “Step-Into”, and “Step-Return”?
Answer:
<p>Task: Practice tracing through the sample program.</p> <p>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.</p> <p>Based on your best understanding of the program, <u>provide a list of methods</u> 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).</p> <p>HINT: Use a combination of “Step-Into” “Step-Over” and “Step-Return”. Use the “Stack Trace” window.</p>
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.

**2. 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 screen capture of original code (with line numbers)

**SCREEN CAPTURE: Original Code with line numbers**

Screen Capture of Original Code  
HERE:

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

TABLE TWO

Line Number	Type of error (compile-time, run-time, or logical)	Description	Correction
...	...	...	...
...	...	...	...

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

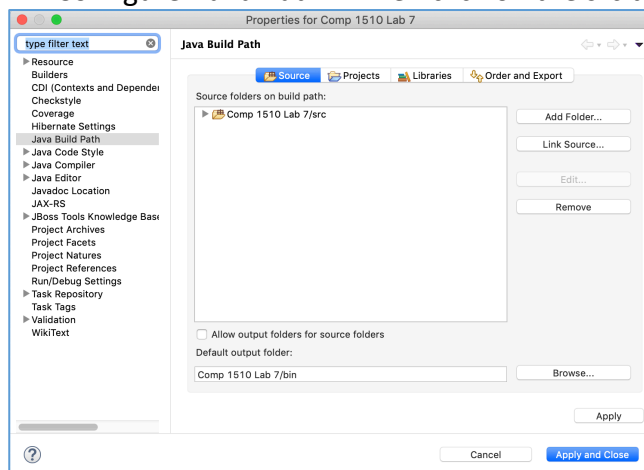
## SCREEN CAPTURE: Fixed code with line numbers

**Screen Capture of Fixed Code,  
and Console Output  
HERE:**

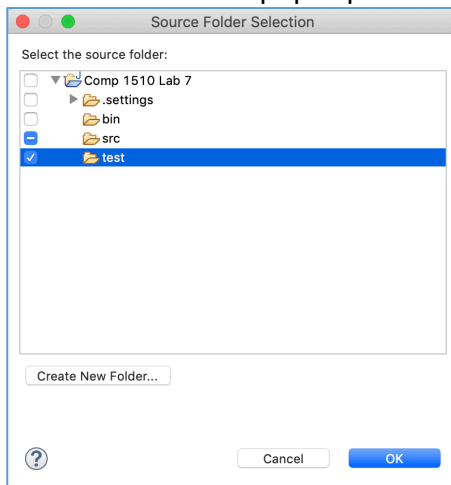
## 3. Writing Unit tests with JUnit.

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. Right click on your project and select Build Path > Configure Build Path ... Then click on the src tab. You should see the following:



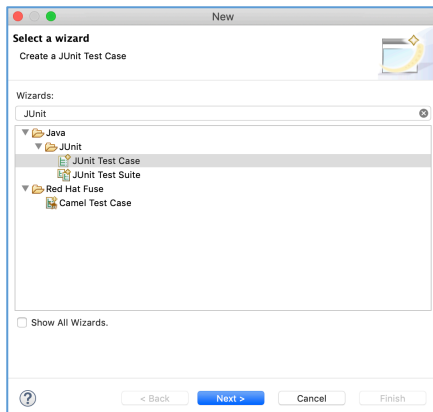
2. Now click on Add Folder, then click on Create New Folder and enter *test* in the Folder name field that pops up. The result should look as follows:



3. 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  **test**. 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).

4. From now, you should put unit test code under the test folder and your regular code under the src folder. Add a package `ca.bcit.comp1510` to the test folder.
5. From the learning hub, download the file `ToTest.java` and copy it into your src folder in the `ca.bcit.comp1510.lav7` package. This will be what we are testing. There are two methods, both finding the largest value of integers.
6. Right click on the package in the test folder and chose New > other and type JUnit in the text field as follow:



Then select Junit Test Case, and in the New Junit Test Case wizard, select New Junit 4 test and enter the test class name (`TestToTest`) and click Finish.

here)' with an unchecked 'Generate comments' checkbox. At the bottom, there is a 'Class under test:' field with a 'Browse...' button. The bottom of the dialog has a '?' icon, '< Back' button, 'Next >' button, 'Cancel' button, and a blue 'Finish' button."/>

7. 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).
8. Now it is a matter of adding test methods. Each method should call one of the methods of the ToTest class and use an assert method (such as assertEquals) to check the correct result. Each method should have a name indicating what is being tested. Start with
  - a. testLargest1\_2\_3
  - b. testLargest2\_3\_1
  - c. testLargest3\_2\_1
  - d. testLargestList1\_2\_3
  - e. and so on.
9. Fix any errors your tests find.
10. 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!

4. You're done! Show your lab instructor your work.