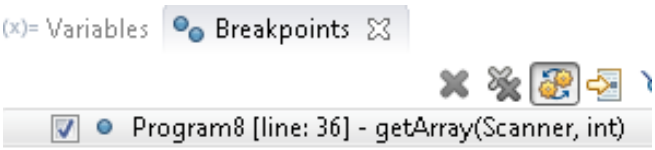


Programming Exercise 10:
Exploring the Debugger

Purpose: Using the debugger tool in Eclipse.
Background readings from textbook: none.

Due date for section 001: Monday, March 28 by 10 am
Due date for section 002: Wednesday, March 30 by 10 am

---

1. Overview

Programs will not compile with syntax errors. Java compilers will point out the location (or the approximate location) of your syntax errors. However, even if a program compiles, it may not be free of errors. Run-time and logical errors can be tremendously difficult to locate and fix. No matter what type of error exists, the process of locating and fixing your errors is known as *debugging*. In order to debug syntax errors, the compiler points out the type of error and approximate location. In order to debug run-time errors, Java usually ends the program noting where the error arose. These are known as Exceptions in Java because we can implement code called exception handlers to fix the problem while the program is executing so that it does not terminate. We do not study exception handling in this course although we will refer to a run-time error caused by an Exception in this lab. Logical errors are very hard to locate. We often have to add output statements throughout our program to figure out what executed and what didn't, and to obtain values of variables. Is there an easier way to locate our logical errors? Yes. In Eclipse, we use the Debugger, a facility that lets us insert *break point* into our code. A break point is a position in your program that, when reached, causes execution to stop so that you can see what the value of the variables are before continuing with the execution of your program.

2. Debugging Support in Eclipse

In order to use break points and explore debugging in Eclipse, we need to switch "perspectives". You can either click on the `Debug` button (see figure 1) or select from the `Windows` menu `Perspective` → `Open Perspective` → `Debug`.

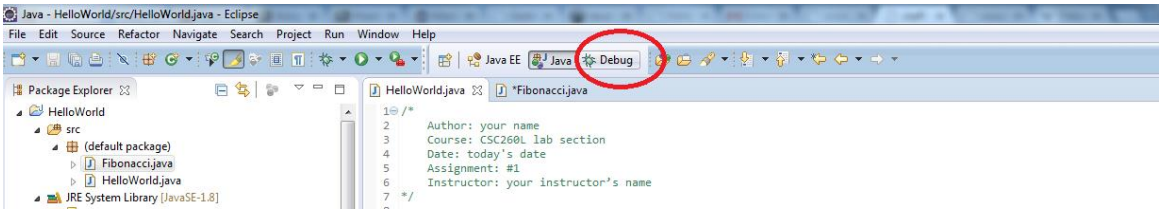


Figure 1: Entering Debug Perspective

In the Debug perspective, you will see two panes added above the editor window, Debug and Variables/Breakpoints. See figure 2.

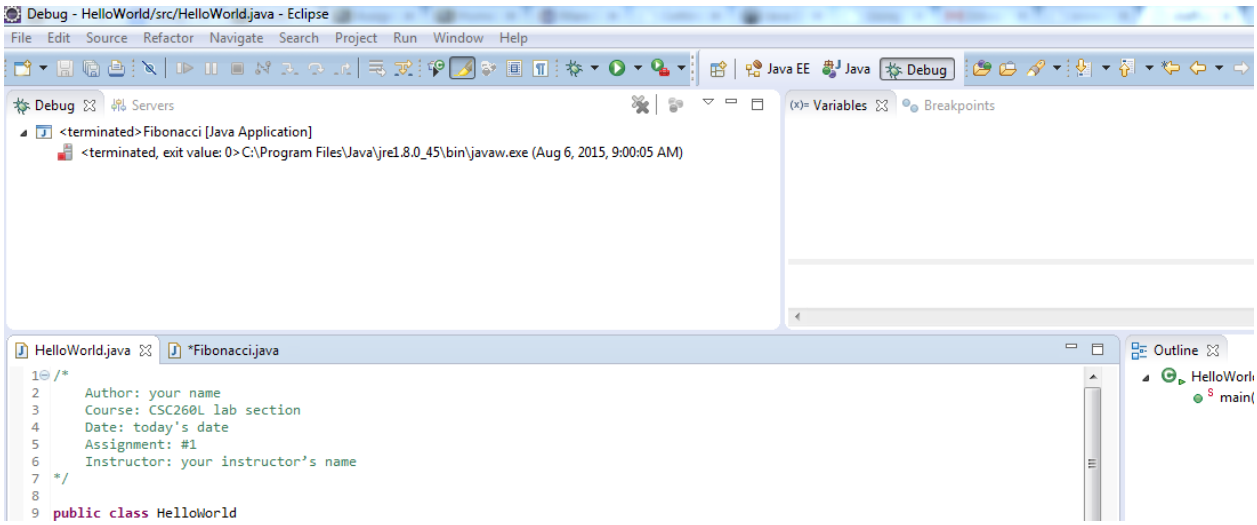
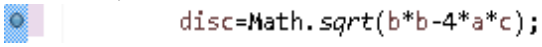


Figure 2: Debug Perspective

In the debugging perspective, you can add break points to your code and inspect the values of variables. To add a break point, double click in the leftmost column of the editor pane on the line where you want to insert your break point. This inserts a blue circle to the left of the line. The circle represents a breakpoint. For instance, we added on to the instruction below.



You will also see a break point entry added to the Breakpoints list in the upper right pane (if you have selected Breakpoints rather than Variables). To remove a breakpoint, double click where the blue circle is. You can also insert and remove breakpoints by right clicking in the leftmost column of the instruction and selecting Toggle Breakpoint from the pop-up menu.

If you run the program normally, the breakpoints are ignored. Instead, run the program by selecting `Debug` from the `Run` menu (or click on the button that looks like a spider  , or press `F11`). When the program reaches a break point, it will pause. You can now inspect what is going on in the program. Under the Debug pane, you will see a "run trace" of the program (what methods are active and the order they were invoked). If you select Variables in the upper right pane, you will see the variables declared to this point of the program and their current values. To continue execution of the program, either click on the Resume button () or select `Resume` from the `Run` menu, or press `F8`. You can halt the program by clicking on the Stop button () or by selecting `Terminate` from the `Run` menu (or `Control+F2`). In resuming, the program will continue until it reaches another breakpoint. If the breakpoint is in a loop or in a method called from within a loop, it will stop every time it is reached. Next, we will put this into practice.

3. Problem: Quadratic Equations

We will write a program that will motivate why we want to use the debugging facilities. In this program, you will compute the roots of a quadratic equation. A quadratic equation has the form $ax^2 + bx + c = 0$ where `a`, `b`, and `c` are coefficients (specific values) and x is a variable. We will input a, b and c from the user and then compute the values of x that make this equation true. These

values of x are known as the *roots* of the quadratic equation and are computed using the following formulas.

$$root_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ and } root_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Notice that both equations require that we compute the square root of $b^2 - 4ac$, which is called the *discriminant* of the quadratic equation. If the discriminant is positive, the equation has two roots. If it is zero, the equation has one root. If it is negative, the equation has no roots (its roots are imaginary numbers). We do not want to take the square root of a negative value as the computer cannot do this. In most programming languages, doing so would cause the program to terminate with a run-time error. In Java, this causes the values NaN (not a number) to be produced. Similarly, we do not want to divide by 0 as this would result in a run-time error in most languages and the value Infinity in Java. We prefer not to output NaN or Infinity as the user may not have any idea why these values arose.

Create a program called Quadratic.java which consists of three methods: main, computeDiscriminant and computeRoot. In main, declare a Scanner, int values a, b, c and doubles root1, root2. Input a, b and c from the user and then have the following three assignment statements:

```
discriminant = computeDiscriminant(a, b, c);
root1 = computeRoot(discriminant, a, b, '+');
root2 = computeRoot(discriminant, a, b, '-');
```

Finally, output the two roots.

The computeDiscriminant method will compute and return the double `Math.sqrt(b*b-4*a*c)`. The computeRoot method will receive the discriminant (a double), the two int values a and b, and a char of either '+' or '-'. Using an if-else statement, determine if the char is '+' or '-' and compute and return either `(-b+discriminant)/(2*a)` or `(-b-discriminant)/(2*a)`. We do not need to do this in methods but are doing so to see what happens in the Debug pane.

After writing the program, save and compile it, fixing any syntax errors. Then, run it using 1, 3 and 1 as the inputs. The outputs should be approximately -0.3819660112501051 and -2.618033988749895.

4. Using the Debugger

Let's add two breakpoints, one at the return statement in `computeDiscrimnant` and one in the if statement in `computeRoot`. Run your program in Debug mode, using the same three input values. You do not need to recompile your program once you have added breakpoints.

As you run it, notice the Debug pane. It lists that you are currently running Quadratic, it tells you your machine's name and it is running a Thread called main. Upon entering the 3 variable values, execution reaches the breakpoint in computeDiscriminant. Now the trace shows us that you are in Quadratic.computeDiscriminant and gives you the line number within the file. In the upper right hand pane, if you have not already selected Variables, do so. You will see your three parameters passed to computeDiscriminant, and their values. In the editor pane, you will see the instruction with the breakpoint is highlighted. To continue execution, click on the Resume button or press F8. Notice in the Debug window, the trace now shows you are in the other method (Quadractic.computeRoot). The variables (parameters) of this method are listed in the Variables pane (in my case, I called them d, a, b, c in that order and they have the values 2.23606797749979, 1, 3, '+' respectively). Resume and

the program will stop there again. This time, the only difference is the value for the character parameter (which is '-' now) and this line is highlighted in yellow. Why? It is showing you that a variable's value has changed since you last reached this breakpoint. In this case, the char parameter changed from '+' to '-'.

In the editor pane, select one of the parameters by double clicking on it. A pop-up window will appear giving you its value. This might be useful if you have many variables/parameters in the Variables listing or that pane is currently showing Breakpoints instead of Variables. Resume again and the program will terminate, showing you the output.

Let's explore now why its important to use breakpoints. Run the program in normal mode (not Debug mode). Enter 1, 2, 3 for the three inputs. You will get as output NaN twice. This means that the roots are not numbers indicating that something went wrong but you aren't given any information about what went wrong. Now run the program in Debug mode, again entering 1, 2, 3. As the program stops in computeDiscriminant, we do not see anything significant, only that it is using a=1, b=2, c=3 to perform the operation, but it has yet to compute the value and return it. Resume. Now we see something more significant. In computeRoot, we see the value of the discriminant parameter has the value NaN. This tells us that in computing the discriminant, something went wrong. We are closer to understanding whats going on but not why. Terminate this run of the program. In computeDiscriminant, let's change the code by breaking it up as follows.

```
int temp=b*b-4*a*c;   // note your variable names may differ
double disc=Math.sqrt(temp);  // disc is my name
return disc;                  // the discriminant
```

Add breakpoints to all there instructions. Save and compile your program (because you changed the code) and then run in Debug mode. Again, enter 1, 2, 3. At the first breakpoint, we have no useful information. Resume. Now we see an added variable added to the Variables list, x is -8. Resume. And now we see the value of the discriminant is NaN. Now we are getting somewhere! If we didn't realize it earlier, taking the square root of a negative number is leaving us with NaN.

Let's try another set of inputs. Terminate this run and rerun the program in Debug mode using 0, 2, 3 as the input. Resume until you reach the computeRoot method breakpoint. You should see that the discriminant is 2.0, an acceptable number. As the breakpoint is at the if statement, we do not see anything particularly troubling. Resume through this and the next pauses to see the output. I this case, we get NaN and -Infinity. Oops. So let's break up this method as well. Rewrite the code as:

```
double num, denom;
denom=2*a;
if(c=='+') num=-b+d;
else num=-b-d;
return num/denom;
```

Remove the current breakpoint and add one to the return statement. Run in Debug mode using inputs 0, 2, 3 and resuming through the computeDiscriminant method. The program pauses at the return statement in the first time you call computeRoot. Notice that both num and denom are 0.0. Doing this division (0.0/0.0) yields NaN. Resume and at the next pause, you will see num is -4.0 while denom is 0.0. Doing this division (-4.0/0.0) yields –Infinity.

Now that we see where the problems arise, we can attempt to fix them. First, if the discriminant is NaN, we want to output that no root exists. Second, if a is 0, we want to output that this is not a

quadratic equation (there is no x^2 term since a is 0). We will change the code in main to handle these circumstances. We compare the discriminant in main to NaN by using

```
if(Double.compare(disc,Double.NaN)==0)
     System.out.println("This equation has no roots");
else // call the computeRoot method twice, output result
```

We handle the division by zero case by testing a as in

```
if(a==0)
     System.out.println("This is not a quadratic equation");
else // call computeDiscriminant, computeRoot, output
```

Modify your program and run it on inputs 1, 3, 1, on 1, 2, 3, and on 0, 2, 3 to make sure the output is correct in all three cases.

4. Experiment On Your Own

You will now write a simple program to test your ability to insert break points to help debug logic problems in your program. Write a new program called Mess. It will consist only of a main method. Declare four variables, `list` which is an int array instantiated to 5 elements, `num` which is an int value storing the number of elements in the array initialized to 0, a Scanner, and `max` which is an int initialized to 0. Once you have declared your variables, write the following while loop:

```
while(num<=5) {
     list[num]=input.nextInt();     // Scanner is called input
     number++;
}
```

Next, write a for loop to iterate from 0 to num and find the largest value in the array, storing it in max. We saw how to do this earlier in the semester (see lab 6, part 5). Finally, output the value of max. Run the program and enter as your values 1, 2, 3, 4, 5, 6. You will get a runtime error (an Exception) upon entering 6. Why? Click on the link that indicates the location of the Exception. Between showing you where the error occurred and the name of the error (ArrayIndexOutOfBoundsException) you should be able to figure out the problem. Fix the problem and rerun the program using as input 1, 2, 3, 4, 5. The output is 5. Run again using inputs 5, 3, 7, 2, 6. Now rerun the program with input -5, -3, -7, -2 and -6. Do you get the right output? If not, why not? In order to fix this problem, add breakpoints to your code and run it in Debug mode, inspecting variable values to see what is going wrong. Fix your program's logic. To finish off this assignment, answer the following questions as comments at the bottom of your source code, or in a separate file.

1. How did you fix the ArrayIndexOutOfBoundsException? Why did you not need breakpoints to help solve that problem?
2. How did you fix the problem of getting the wrong max value when you ran the program on input -5, -3, -7, -2, -6? Specifically, where did you insert your breakpoints (on what instructions) and what was the problem?
3. In general, what does the debugger allow you to do? Under what circumstances do you plan on using the debugger? In particular, what types of errors will it help you to resolve and fix?
4. What use is the run trace in the Debug pane? That is, what information is displayed there?
5. In general, where would you plan on placing breakpoints when debugging code? That is, in what types of instructions?

Submit either by hardcopy or email your source code for Mess and the answers to the above questions. Do not submit any output from this program. Do not submit your Quadratic program.