

# Lab 5: Loops, ArrayLists, and Arrays

## Before You Begin!

In this lab, you will be learning about and working with arrays and ArrayLists, two extremely convenient ways of storing data in Java. The [reading](#), [mini-assignment](#), and lectures on this material will be really helpful. If you're at all confused about the content of those materials, review with your partner before beginning!

## Pair Programming:

As with previous labs, be sure to practice good pair programming conventions. One person in the group should be the driver, while the driver is the one typing. All members of the group should be discussing solutions to the problems and reading along as you go. Be sure to switch drivers approximately every fifteen minutes.

## Debugging with Eclipse

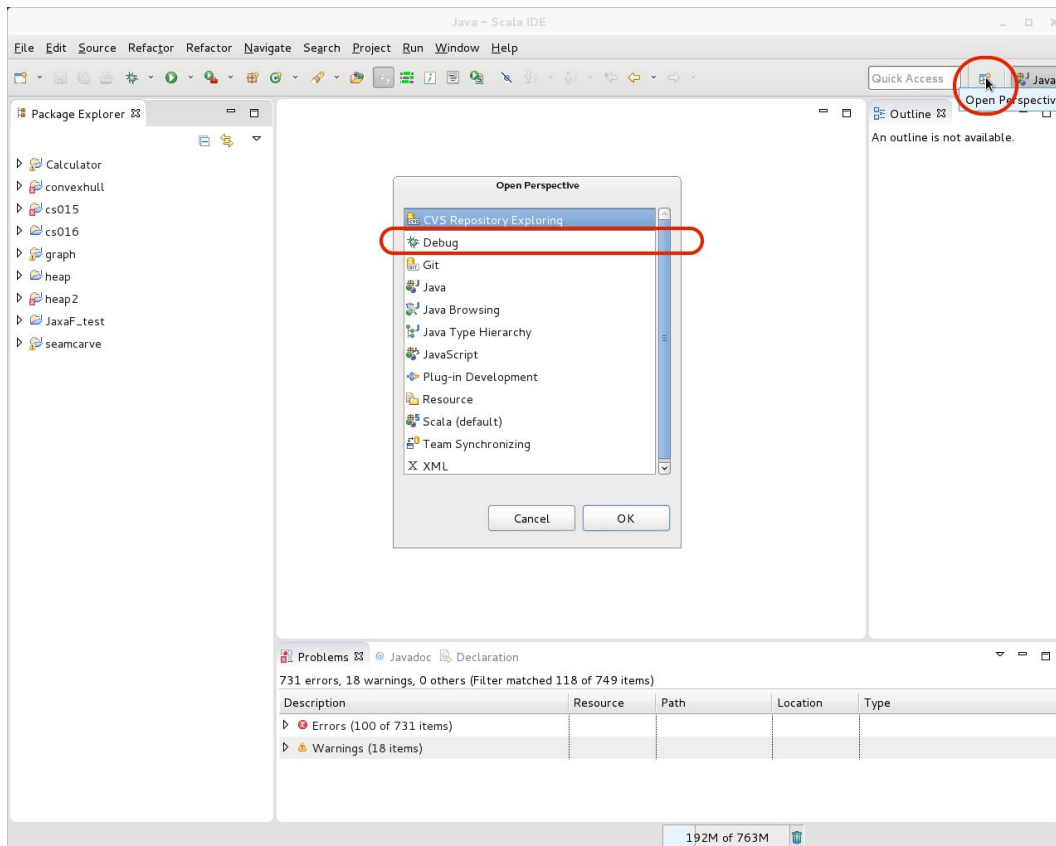
The Eclipse Debugger is designed to give you tools that will make debugging your programs much easier. Rather than writing 20 print lines and running the entire program, the Eclipse debugger allows you to accurately inspect different portions of your code *while it's running* at your discretion.

*Read through this section carefully with your partner, making sure you each understand what is happening. This information will be vital later in the lab!*

## The Debugger Perspective

When you actually use the debugger in Eclipse, you will have to enter the Debugger Perspective to access the Debugging tools. There are multiple ways to enter the Debugger Perspective; we'll cover one method now, and another later.

- Click on the circled button (in the screenshot below, in the upper right corner of your screen), and then select Debug.




- When you are prompted by the `Confirm Perspective Switch` window, click “yes”.

**Note:** If you have used the debugger perspective (and in the future), there will simply be a button that says *Debug* instead of *Open Perspective*

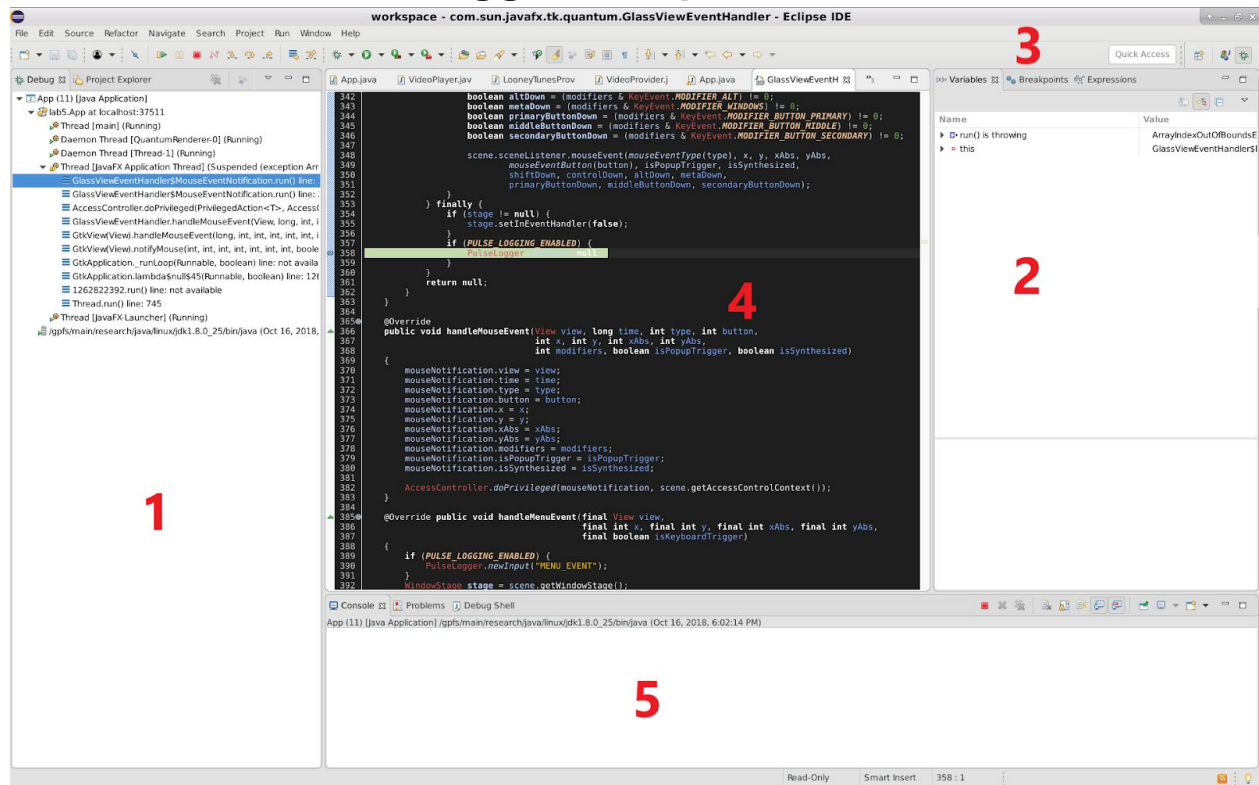
After confirming, your window will rearrange and have a spiffy new look. This means you have entered the Debugging Perspective.

You can switch back and forth between the debugging perspective and the normal perspective by toggling the buttons on the upper right.

To run code in normal mode, you would originally click the green “run” button (  ). To run

code in debug mode, you must click the “bug” button (  ).

# Overview of the Debugger Perspective

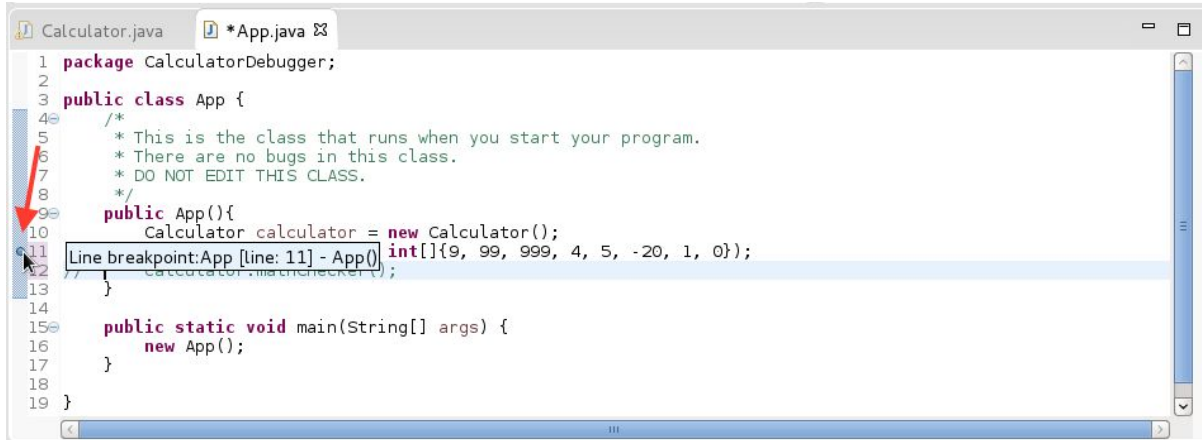


1. **Call stack:** This shows where your program is in its execution. It is the same stack trace you would see in an error message in the terminal!
2. **Variables:** All of the variables (instance and local) accessible from the current scope.
3. **Breakpoints:** A list of all the breakpoints you have set, which you can toggle on and off. Click this tab to view them.
4. **Code:** Your code! Exactly what you see in the regular Java perspective, but smaller.
5. **Console:** The same console that you normally see in the Java perspective.

We will examine some of these items in more detail.

## Breakpoints

A breakpoint is a user-selected line of code that the program will stop at (once that line is reached). Breakpoints allow you to pause your program mid-execution and look directly at the value of your variables. This eliminates having to constantly print out variable values.



Double-click right next to a line to put a breakpoint there. When in Debug mode, this breakpoint will pause the program right **before** the contents of the line are executed. Double-clicking again will remove the breakpoint.

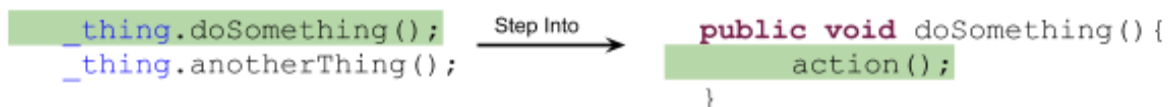
## Running, Pausing, Stopping, and Stepping

Eclipse also lets you step through the execution of your program line-by-line while still allowing you to follow the flow of your program from one method to the next. There are four useful step functions: step into, step over, step return, and drop to frame. The buttons for each of these are in the debug menu bar.



1. **Step Into:** If there is a method call in the current line, you can “step into” the method being executed, which means the next line you will investigate is the first line of the method you are stepping into (wherever that method is defined).

Example: If you have a method called `doSomething()`, and you “step into” the line `_thing.doSomething()`, you would be taken to the class and line where `doSomething()` is defined.



**Beware!** If you “step into” a line that calls a method defined by Java, you will be taken to strange, foreign classes that you may not care to see!

2. **Step Over:** This function allows you to move down your lines of code, one line at a time, without “stepping into” the lines. This is helpful for walking through the code and inspecting variables when you aren’t interested in the inner workings of a method.

Example: If you "step over" the line `_thing.doSomething()`, you would simply be taken to the line that follows `_thing.doSomething()`.



3. **Step Return:** Use this function when you are done analyzing the variables in a method call and want to return to the line that called it. Essentially, it just advances to the current method's return statement.
4. **Drop to Frame:** This is a tool that will allow you to assess the state of the program when the current method was called. In more casual terms, Drop to Frame essentially just lets you do a "do-over" while debugging by returning you to the top of the method that you are running and when you return to the top, the state is the same as when you first entered.

## Coloring Arrays

You're going to initialize and populate a few `javafx.scene.paint.Color` arrays of specified dimensions, using different colors to form patterns. You'll also have to write code to allow you to visualize the arrays you've created.

1. Run `cs0150_install lab5` to install this lab's stencil code. Open Eclipse and check out the contents of the `lab5` package (refresh Eclipse if you don't see it). For the first exercise, you'll be working within the file `ArrayBuilder.java`. Open it up now.
2. The `ArrayBuilder` class contains one method called `displayArray` followed by four methods which will each build a 2D array of colors. One method, `buildStripeArray`, has been written for you. Run the program by right-clicking on `lab5's App.java` class and choosing "Run as >> Java Application". Click the Stripes button once the window appears. Nothing shows up!

This is because the `buildStripeArray` method only initializes/returns an array of colors. It does not actually create `Nodes` that can be added to a parent `Pane`, or add those `Nodes`. Remember - arrays allow you to logically organize objects within your program. They don't have anything to do with JavaFX graphics.

3. To display this array we need to convert it into an array of rectangles with colors corresponding to the array created in `buildStripeArray`, and then add these rectangles to a `Pane`. You will need to complete the `displayArray` method in order to accomplish this. We have already handled converting an array of `Colors` into an array of `Rectangles` for you.
  - o In the `displayArray` method, clear all of the children of `parentPane` to get rid of any previously displayed arrays. Remember, you can get the children of a `Pane` by calling the `getChildren()` method. This returns a `List` object. Not

sure how to clear a `List`? Check out the [Javadocs!](#)

- Loop through the array of `Rectangles` passed as a parameter, adding each one to `parentPane`. To do this, copy the following code:

```
for (int i = 0; i <= rects.length; i++) {  
    for (int j = 0; j <= rects[0].length; j++) {  
        // Code to add to the pane goes here!  
    }  
}
```

**Note:** There are two (intentional!) bugs in our code. Running what you just typed as is will result in an error! Use your knowledge of the Eclipse debugger to find out why. Some hints:

1. One error is located in the `displayArray` method you just wrote. The other is in your `buildStripeArray` method. The error in `buildStripeArray` will occur *before* the `displayArray` error.
2. You might not get a stack trace, and instead be taken to a build-in Java class you don't recognize. This is OK! You can click the continue button at

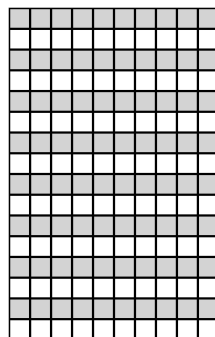


the top of your screen ( ) and Eclipse will print a stack trace.

3. The error is related to your loops. You'll probably want to set a breakpoint there!
4. Remember that the debugger allows you to view the values of variables — including *i* and *j* from your loops....

Keep track of what you fixed. You'll need to show a TA you can use the debugger in order to get checked off!

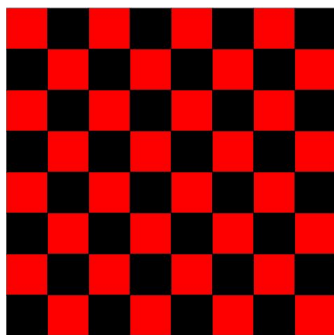
4. Once you've found the bug, in order to check that your `displayArray` is working, run the program to check that the stripe array returned by `buildStripeArray` is displayed correctly. Once the window appears, click the "Stripes" button (under the "Arrays" tab). You should see this array:



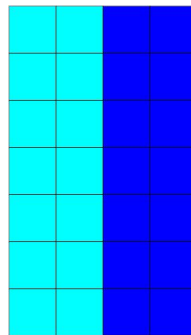
5. Fill in the remaining three stencil methods, `buildCheckerArray`, `buildTwoColorArray`, and `buildDiagonalStripeArray`.
  - In each method:

- i. Create an array of `Colors` of the dimensions specified in the method comments.
  - ii. Next, loop through the array you've created, figuring out the appropriate `Color` at each index and storing it in the array.
  - iii. Finally, `return` the array you've created.
- If you're having trouble getting the expected results, try adding a breakpoint to the start of the method that's giving you issues, and analyze the method step by step to make sure that the method is doing what you expect it to be doing! Remember that the debugger can be used to view the values of variables - how might this be helpful in analyzing your loops?

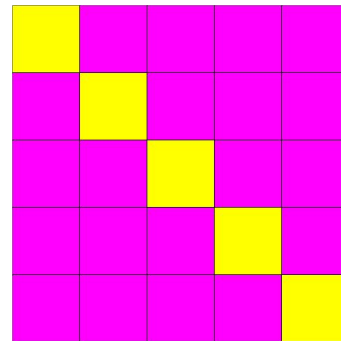
Expected Results:



Checkerboard



Two-color



Diagonal

6. Run the program and test each method by clicking the corresponding button in the GUI, which will call your method and visualize the array of `Colors` it returns. The images above show what each array should look like-- when your arrays match them, move on to the next part of the lab.

**Check Point 1:** Call over a TA to check your arrays! This would be a good time to switch partners if you haven't already.

## Practice with ArrayLists

You're going to fill in a series of methods, each of which performs a particular operation on an `ArrayList<Color>` called `_colors`. Again, you'll first write code to visualize the `ArrayList`!

1. Open up the file `ArrayListBuilder.java`. The `ArrayListBuilder` class contains several stencil methods, each of which should perform an operation on the `ArrayList<Color>` instance variable called `_colors`.
2. In your `displayArrayList`, fill your `ArrayList` with two pink rectangles. This way, you have something to show when your program first runs!

3. Again you will need to fill in the method `displayArrayList` before anything will show up on the screen. Clear the `parentPane` and add all the elements of `rects` to the `parentPane`.
  4. Run the app and navigate to the “ArrayLists” tab. Each of the buttons at the bottom of the interface calls the corresponding method in the stencil code. Play around with the “Add Pink” and “Clear” buttons (we’ve filled in these methods for you-- the other buttons won’t do anything until you fill in their methods)! If you’ve implemented `displayArrayList` correctly, you should see the visualization of `_colors` update as elements are added and removed.
  5. Fill in the remaining stencil methods according to the descriptions in the method comments. You can reference the Javadocs for a full description of the `ArrayList`’s methods. Run the app and play around to test your work.
- 

**Check Point 2:** Call over a TA to check you off!