JavaDocs & Debugging (Part 2)

The best place to look for how to use the classes already implemented by Java are the **Java Docs**. You might have heard the word mentioned before, but what are these mysterious things called "JavaDocs"?

First, let's talk about Java API. Java **API** (Application Programming Interface) is a collection of built-in classes in Java that provide commonly needed functionality for programmers like yourself (like java.awt.Color or javax.swing.JPanel). After all, do you really want to program how a frame and buttons work? Don't you have better things to do with your time? Java API is "support code" provided directly by Java and available to anyone who uses Java, its "magic".

The sheer number of methods and classes that Java API provides for you is staggering. How are you supposed to keep track of them all?

The JavaDocs!

JavaDocs is the documentation of the entire Java API. Treat it like a user's guide to Java libraries. JavaDocs tell you which classes you can use and how you should use them, as well as brief description of what all the methods in each class and what they do. JavaDocs also include information about the inheritance structure of classes and much, much more...

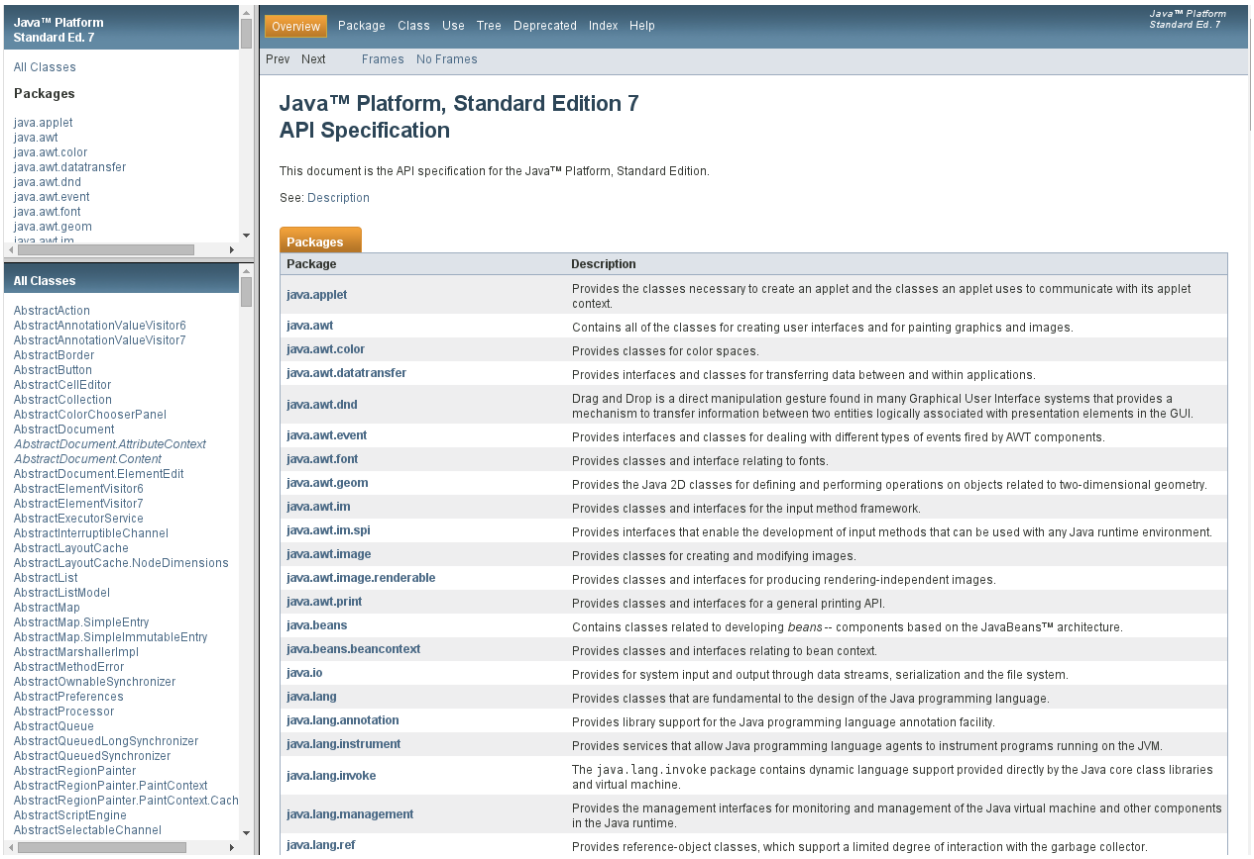
You don't believe me do you? How can one set of documentation provide all the answers you might want? Let's run through a few scenarios:

- Do you want know what parameters the constructor for JSliders take?
    - *Look at the Java Docs*
- Do you want to know what the subclasses of java.awt.geom.RectangularShape are?
    - *Look at the Java Docs*
- Do you want to know what method to call to set radio buttons?
    - *Look at the Java Docs*
- Confused about life in general?
    - *Look at the Java Docs*

Where are these magical JavaDocs you ask?

We have conveniently provided a link to them on the CS15 website.
Simply go to the CS15 website >> Documentation >> JavaDocs.

A window that looks like this will open:



There are three parts to the Java Docs:

1. **Package List**: Lists all the available packages, for example: javax.swing and java.awt.

2. **Class List**: Lists all the classes contained within a particular package, for example: if you are select javax.swing you would find JButton and JFrame classes listed here among many others.

3. **Class Details**: When you click on a class from the class list it will display all information specific to that class.

Navigation

You can navigate to the information page for a class that you want by:

1. Selecting the appropriate package in the 1st panel.

2. Selecting the needed class name from the 2nd panel.

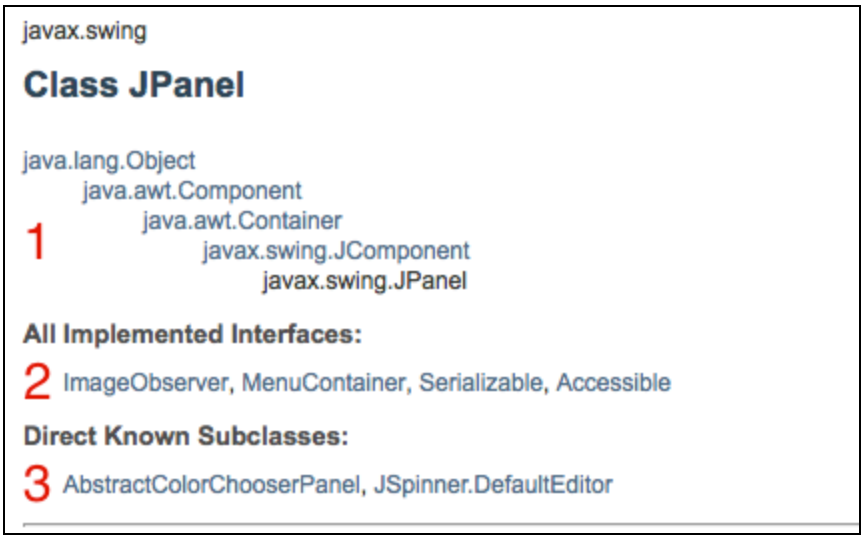
3. Details of that class will appear on the 3rd panel.

Now, this will require some prior knowledge, that is, the class's package, but you should have no problem figuring out where the classes you'll use for CS15 are--we constantly reference them in our lecture slides. For example, javax.swing.JPanel gives the information that JPanel is in the javax.swing package.

Alternatively, if you search for the name of a Java class in Google, the corresponding Java Docs page will often be one of the first results.

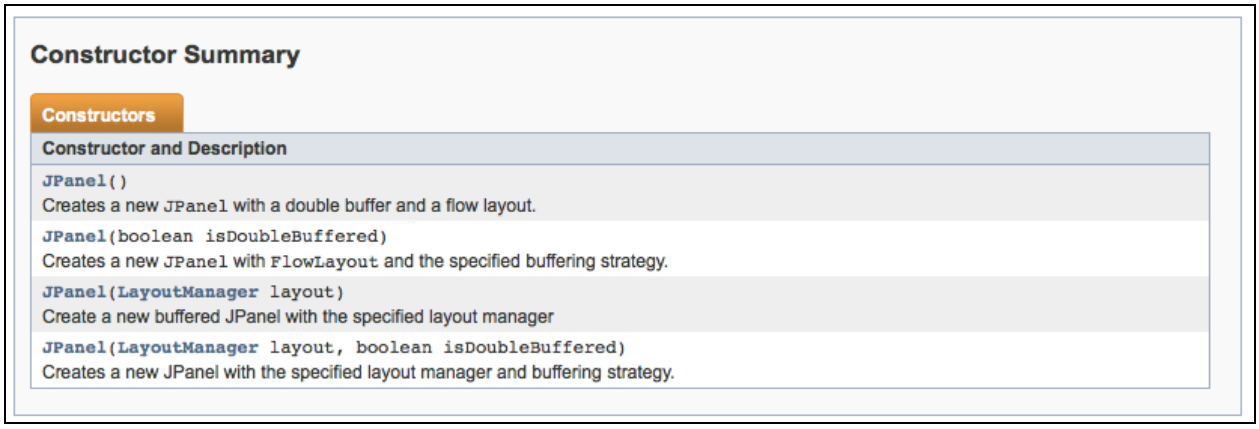
Another useful technique for navigating the JavaDocs is the "Find" feature provided by Firefox and other internet browsers. This will be particularly useful when trying to find a class description in the Class Details.

The real usefulness of the Java Docs is in the Class Details panel.



1. Provides the **inheritance tree** for the class.
2. Provides the **interfaces** the class implements.
3. Provides a list of all known **subclasses** of the class.

After those nuggets of knowledge comes a summary of the class and then

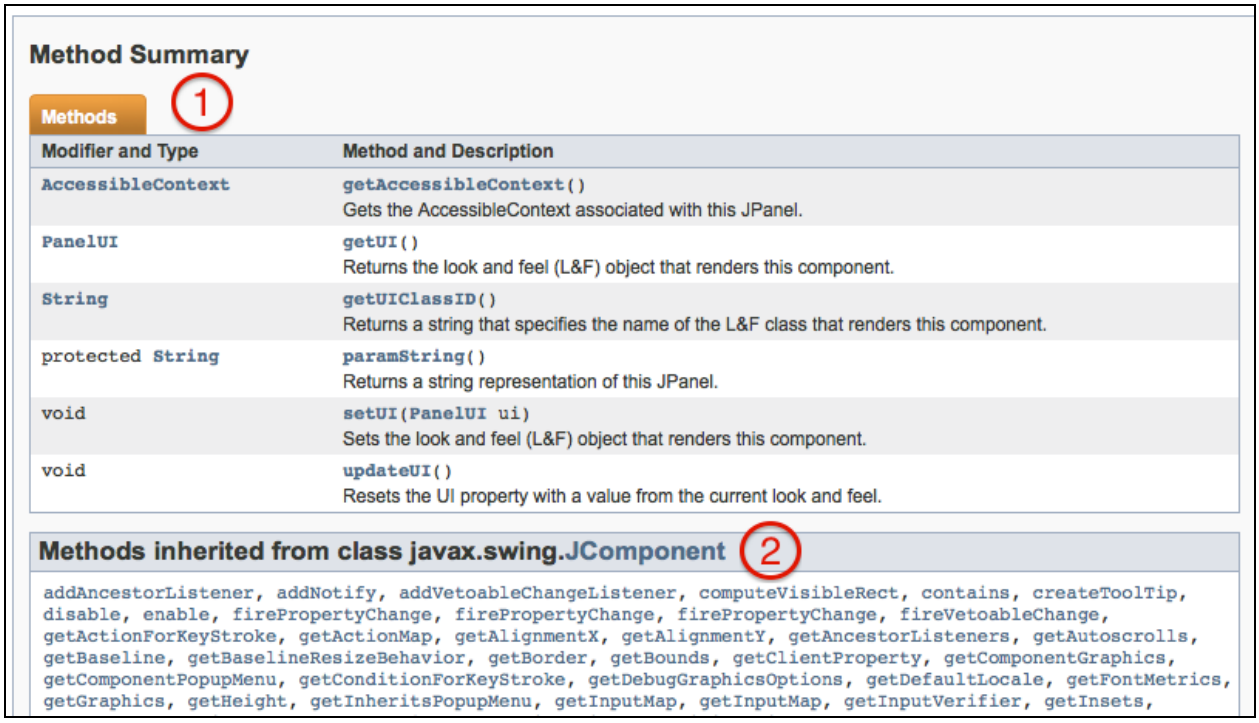


Constructor Summary

Constructors

Constructor and Description
`JPanel()` Creates a new `JPanel` with a double buffer and a flow layout.
`JPanel(boolean isDoubleBuffered)` Creates a new `JPanel` with `FlowLayout` and the specified buffering strategy.
`JPanel(LayoutManager layout)` Create a new buffered JPanel with the specified layout manager
`JPanel(LayoutManager layout, boolean isDoubleBuffered)` Creates a new JPanel with the specified layout manager and buffering strategy.

Constructor Summary: lists all of the constructors.



Method Summary

Methods 1

Modifier and Type	Method and Description
`AccessibleContext`	`getAccessibleContext()` Gets the AccessibleContext associated with this JPanel.
`PanelUI`	`getUI()` Returns the look and feel (L&F) object that renders this component.
`String`	`getUIClassID()` Returns a string that specifies the name of the L&F class that renders this component.
`protected String`	`paramString()` Returns a string representation of this JPanel.
`void`	`setUI(PanelUI ui)` Sets the look and feel (L&F) object that renders this component.
`void`	`updateUI()` Resets the UI property with a value from the current look and feel.

Methods inherited from class javax.swing.JComponent 2

`addAncestorListener, addNotify, addVetoableChangeListener, computeVisibleRect, contains, createToolTip, disable, enable, firePropertyChange, firePropertyChange, firePropertyChange, fireVetoableChange, getActionForKeyStroke, getActionMap, getAlignmentX, getAlignmentY, getAncestorListeners, getAutoscrolls, getBaseline, getBaselineResizeBehavior, getBorder, getBounds, getClientProperty, getComponentGraphics, getComponentPopupMenu, getConditionForKeyStroke, getDebugGraphicsOptions, getDefaultLocale, getFontMetrics, getGraphics, getHeight, getInheritsPopupMenu, getInputMap, getInputMap, getInputVerifier, getInsets,`

1. **Method Summary:** lists all methods written in this class (does not include inherited methods).
2. **Inherited Method Summary:** lists all methods that this class inherited from its superclass.

Check Point 1: Learning about JavaDocs

1. Open up a blank text file in Sublime (type ‘sublime &’ in a shell).

2. Navigate to the JavaDocs from the cs015 homepage.
3. Click the `javax.swing` package name in the Package List panel. The Class List panel of the Java Docs will now display only classes found in the package `javax.swing`!
4. Now select `JPanel` from the list of classes. The Class Details section now displays the information for `JPanel`'s class!
5. In the text editor, write the answers to the following questions:
    a. What are the signatures of `JPanel`'s four constructors?
    b. What is the return type of the method `getActionMap(...)`?
    c. From what class does a `JPanel` inherit the method `setFocusable(...)`?
    d. What is the type of parameter that the method `addMouseListener(...)` expects?

Debugging (Part 2): The Eclipse Debugger

In the Debugging (Part 1) lab we learned how to debug using Eclipse. As you may know, debugging isn't always clear cut and sometimes bugs can be hard to spot. Lucky for you, Eclipse has a powerful debugger that allows you to examine the state of your program as it's running. You can stop the program when it gets to certain lines of code, look at the values of all the existing variables, and even step through your code line by line.

Though printlines will usually suffice for fixing your average NullPointerException, the debugger will save you bajillions of printlines when it comes to fixing more insidious bugs, especially as your programs grow more complex.

So without further ado, we present *drumroll* the Eclipse Debugger.

Goal: Learn to work with the Eclipse Debugger.

Getting started with the debugger

For this lab, we will be using a new program to demonstrate the awesome power of the debugger.

Check Point 2: Getting Started

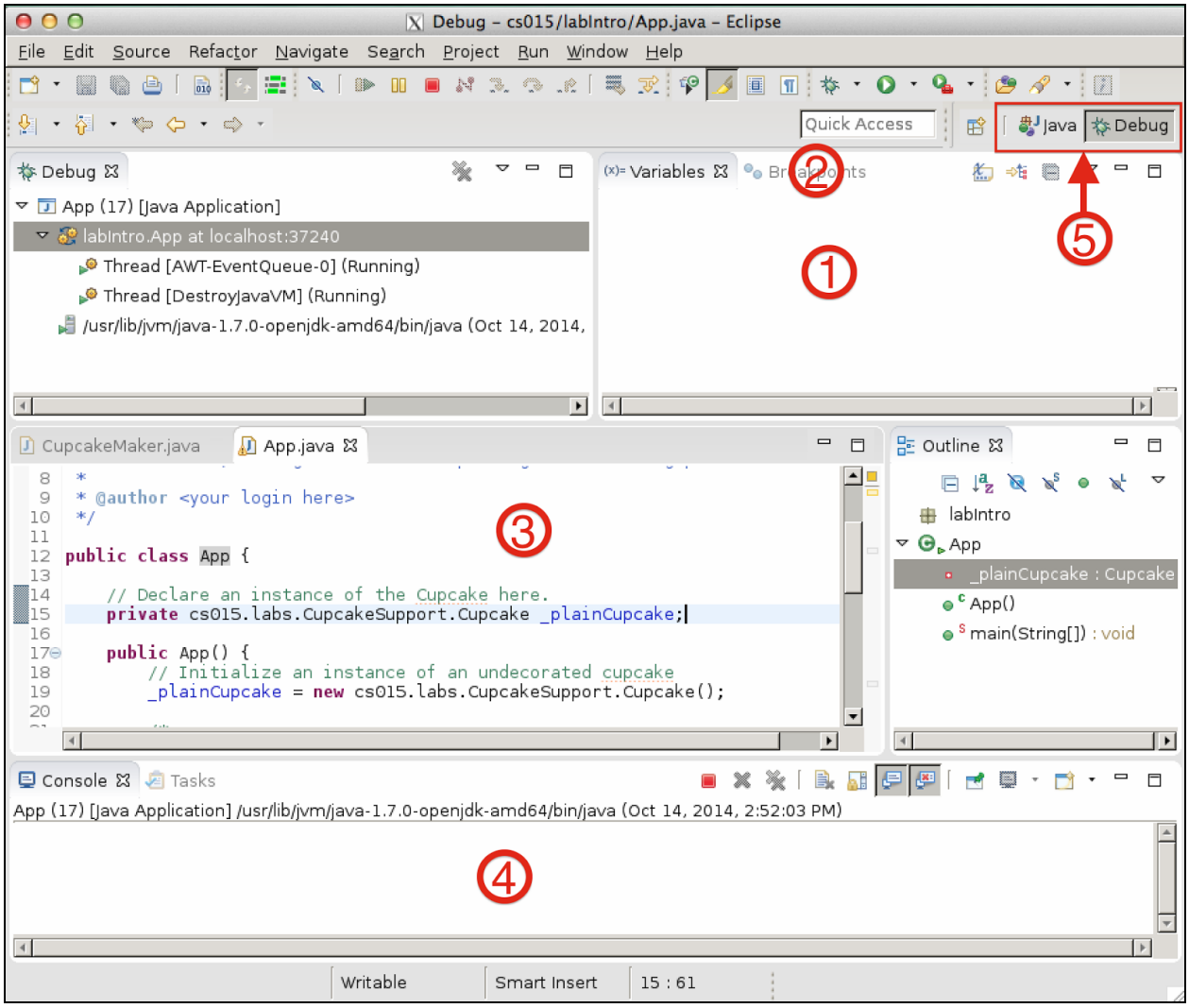
1. Run `cs015_install labDebugging` to install the code for this lab.
2. In Eclipse, right click on your CS015 project and click "Refresh" (or press F5). You should see "labDebugging" appear in the list of packages.

Debugging in the Debugger Perspective

The Debugger Perspective

When you actually go to use the debugger in Eclipse, you will have to enter the Debugger Perspective. To do this, you just have to right-click on your App class and click “Debug As…” >> “Java Application”. When you are prompted with the following window, click “yes”.

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



As you can see, there are a lot of modules in the Debugger Perspective. Below we have explained what we feel are the most important modules:

1. **Variables:** All of the variables (instance and local) accessible from the current method.
2. **Breakpoints:** A list of all the breakpoints you have set, which you can toggle on and off.
3. **Code:** Your code! Exactly what you see in the regular Java perspective, but smaller.
4. **Console:** The same console that you normally see in the Java perspective.
5. Switch between the Debugger Perspective and the Java Perspective.

Now that we know a little bit about the Debugging Perspective, let's learn about the tools we can utilize.

Breakpoints

For most people, the most common feature you will use in the debugger are 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 and spending time on interpreting the mass of values printed.

This is what a breakpoint looks like:

You see that there is a blue dot in the column to the left of the line numbers--this is a breakpoint. You can toggle (add or remove) breakpoints by double clicking on a line number or, more formally, by right-clicking the line number and clicking “Toggle Breakpoint”.

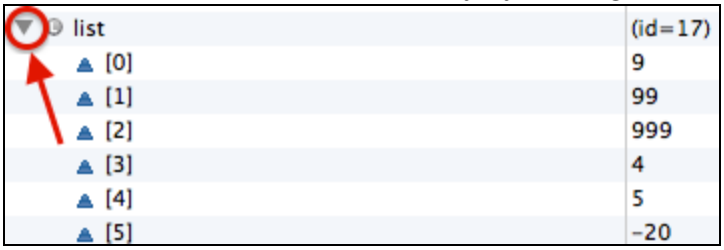
In the example above, the breakpoint occurs on line 10, so when we enter the Debugging Perspective, the program will pause **before** it begins line 10.

So let’s do a little practice with breakpoints!

Check Point 3: Breakpoints and the Debugger Perspective

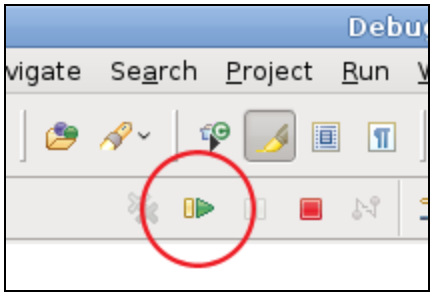
1. In Calculator.java, add a breakpoint to line 139 (`int length = list.length;`)
2. Now run the program in debug mode. Right-click on App.java and select “Debug As… >> “Java Application”.

   Hint: In the future, you can just click the green bug next to the green run button.
   Remember: Confirm the perspective switch into the Debugging Perspective.
3. Look at the Variable Module, in a text editor (Sublime) answer the following questions
   What variables are accessible at this point in the program?
   What are their values?
   Why are some variables unlisted?
4. You can look at the element values of the array by clicking the triangle next to list:



5. Look at the Debug window and write down the stack trace in your text document.

6. Unpause your program by clicking the little green ***resume*** arrow in the Debug menu bar.



(Now you should see the sorted string printed to the console)

5. Add another breakpoint at line 144 (`this.swap(...){`)

6. Run the debugger again.
Your program should suspend on line 139 like last time.

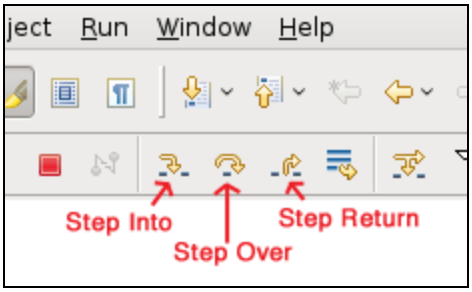
7. Now press the resume button. ( )
It should suspend again! But this time on line 144.

Note: "Resume" always runs your program until it hits another breakpoint. If there are no breakpoints left, your program will just run through to the end.

8. In the same text document, answer the following questions:
    What variables are accessible now (Line 144)?
    What are their values?
    What happens if you try to add a breakpoint to a line without any code on it? Why?
9. Click the Stop (terminate) button, which is the red square next to the play button, to terminate the program.

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 three usable step functions: step into, step over, and step return. The buttons for each of these are in the debug menu bar.



Step Over: This function allows you to move down your lines of code, one line at a time. You can use this to monitor how a variable's value changes after any method calls in a given line of code.

Step Into: If there is a method call in the current line, you can “step into” the method being executed, which means you will jump to wherever the method is defined in the code.

For example, if you have a method called doSomething(), and you "step into" the line where this method is called (i.e. _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 neither need to nor want to know about!

Also, keep in mind that if you try to step into a line that has no method calls, the debugger will simply step over said line.

Step Return: The last functionality you'll learn today is the step return function. Use this function when you are done analyzing the variables in a method call and want to return to the line that called it (similar to step 6 in the previous checkpoint). Essentially, it just skips to the current method's return statement, hence "step return." Try this out:

Check Point 5: “All together now!”

1. Run the program in debug mode.
2. Use the "step over" button to move down until you reach the breakpoint at line 144.

   Note: this may take a lot of “step overs” because the breakpoint is only reached once the `if` statement containing it is satisfied. (You can expedite this process by clicking resume)
3. On line 144, click the "step into" button, which will take you to the method declaration `swap(...)`.
4. Take a look at the Variables module. **Write down in the text document the values of list (its ID number), a, and b.**
5. Notice that `swap(...)` has now been added to our stack trace in the Debug window.
6. Use step return to go back to where you came from at any point while in `swap(...)`.

   Note: Once you reach the return statement, clicking step into or step over will take you back to where you came from too.
7. Keep clicking resume (reminder: ) and watch as the list is being sorted.
8. **Show the text document to a TA.**

Great! Now that you're familiar with the debugger, let's actually debug something.

Using the debugger should eliminate the need for debugging with printlines. How clean and convenient!

Check Point 7: Debug a Program

- Temporarily exit the Debugger perspective.
- In the App class, uncomment line 12 (`calculator.mathChecker();`)
  In the App class, comment out line 11 (`calculator.sortList(...);`)

  Hint: hit "ctrl" + "/" to quickly toggle between commented and uncommented
- Fill in lines 31 - 39 in the `Calculator` class. We have provided detailed comments in the program as to what "TO DO".
- Run the program Calculator by right clicking App.java and then selecting Run As >> Java Application
- You should see that the Eclipse console has popped up with a runtime error. Using th debugger, fix the errors and get the code to run successfully. Make sure to save frequently.
- When you've solved all the problems with your code, you should see `'Finished check. All methods are mathematically correct.'` printed to the console.
- **Once your program is working, show a TA. This will get you checked off for this week's lab.**

Conclusion

Debugging: Completed.

If you want to exit debug mode now, you can click on Window >> Open Perspective >> Java, or click the arrow in the upper right and click Java.

Now that you've learned all about run-time errors, printlns, and the Eclipse debugger, you're fully equipped to tackle your bugs head on. You can spend more time working on your code and less time losing your sanity over a NullPointerException! And being able to debug your code efficiently is a skill essential to any programmer's skillset!

Remember: TAs reserve the right to turn you away from hours if you have not attempted to debug your code before asking for help.