CIS 121 Tutorial

Using the Debugger
Learning how to properly debug your code
Edited by Ziad Ben Hadj-Alouane

As coding projects become much more complex, you might want to think about debugging your code properly. While using print statements is a valid way of finding easy bugs, it is extremely hard to fix many other bugs due to the complexity of the code. To solve this issue, you should familiarize yourself with the debugger, a tool not only available within Eclipse, but in almost all Integrated Development Environments. In short, learning to use the debugger is important if you’re serious about making functioning code.

NOTE: We expect you to know how to use the debugger. We know that some students give up on debugging REALLY fast and resort to Office Hours for any sort of problems in their code. That is legitimate, but more than once you’ll find out that a TA will ask you to use the debugger, which ends up fixing most if not all issues. In short, please learn how to use the debugger, use it rigorously, then ask questions!

In-depth video tutorial
We made a video tutorial that can be found here that explains all the basics of using the debugger.

Alternative Guide
If you wish to review the steps on paper here they are here:

1. Setting up a Breakpoint
   a. Make sure to place breakpoints in places you think you want to examine. This might be inside a test case, inside a class method, etc...
   b. To set up a break point, double click on the sidebar

2. Running the Debugger
   a. After you have set-up some break points, you can run the debugger by either clicking on the Bug icon, or by clicking on Run → Debug
   b. Once you run the Debugger, the runtime machine will stop executing at the first breakpoint it encounters

3. Stepping through your code
   a. Once you stopped at a breakpoint, you can do the following:
      i. Step Into: goes inside an instruction (for example a function, so you can examine what that function does)
      ii. Step Over: executes the current instruction and jumps to the next instruction
      iii. Step Return: pops back up a call stack. For example, if you are inside a function bar() that was called within foo(), you would return back to where bar() was called. Keep in mind that this will first execute the remaining of bar().
iv. **Resume**: executes everything up until the next breakpoint, or completes your program

v. **Terminate**: terminates the execution cleanly

4. **Checking the Variables view**
   a. To check how your variables are being updated, it is handy to have the Variables view ready on the Debugger. When you call the debugger, the Variables view should already appear. If that is not the case, click on **Window ➔ Show View ➔ Variables**
   b. In the Variables view, you can see the fields on the current object (this), the global variables, any local variables, etc... Use this to make sanity checks!

5. **Checking the Call Stack**
   a. The Callstack is the same as the Debug view. You can find this the same way you open up the Variables view.
   b. You probably do not need this as much, but it will become handy in some cases (example: calling functions recursively). The call stack is meant to show you the sequence of functions currently called in order. This way, you can make sure the correct methods are called at the correct line numbers
6. **Going back to Java View**
   a. Once you're done debugging, you should terminate the debugger, then go back to the Java view (top right button)

If you have any questions, do not hesitate to post questions on Piazza or go to TA Office Hours. Please note again that we expect you to learn how to use the Debugger. Going to Office Hours without having thoroughly debugged your code and asking for help immediately is not a good way to learn how to program. Debugging is a skill, and this tutorial offers everything you need to know about it (for this class at least!)