# Debugging (aka “oh my god nothing works”) guide

So your code doesn’t work or it’s failing tests. Well there are a few ways to help you debug your code, some of which can be more helpful than others.

## Understanding the Problem

It is very hard to debug code efficiently if you don’t understand (a) what the code is supposed to be doing, and (b) what the code is actually doing. Being clear on these two points are the essential first steps when debugging.

Both of these tasks are best done with the aid of a **concrete example**. Not a “yeah I have a rough example in mind”, but a specific example that you have written down or drawn on a piece of paper. This step is so useful that we will ask to see the example you are working with if you ask us for help.

## Work Through a Concrete Example by Hand

A good example will have just enough structural complexity to exercise your code, with data values that are easy for you to work with in your head. If you are debugging a problem with lists, for example, a random list like `[56, 799, 21, 0, 53]` is likely less effective than a simple `[0, 3, 1]`. For recursive data types (such as trees), work with a nesting depth of 3 (it is easy to write seemingly correct code on two nesting levels that breaks at a third level).

**Before you look at the code**, make sure you are clear on what the code should do when run on your example:

- What output do you expect?
- If your code builds an intermediate data structure (like a hashmap or list), what should that look like for this specific example? Write or draw that intermediate structure as well.

*If you can’t articulate these two steps, you don’t understand the problem sufficiently, and you will waste time trying to debug your code.*

**Now, step through your code, either by hand or by using the IntelliJ Debugger (see below)**. Trace through the steps of your code, tracking values of variables or intermediate data structures as you go. Again, you should know what you want the code to do before you get to this step. When you find something that doesn’t make sense, you are closing in on the bug.

As you go, remember that **test cases are a valuable aid in debugging**. Let's say you are hand-tracing your code and get to a call to another method to do part of the computation. Do you assume that that method returned the right answer, or do you step into that as well? If you've already run a test case on that method based on your concrete example, you don't have to guess! *Write tests strategically to help narrow down your search for a bug.*

## Getting Insight from the Error Message

Another valuable debugging skill is learning to associate certain kinds of error messages with certain kinds of problems. This can help you figure out which lines of your code might be most problematic. For example:

- A ***Null Pointer exception*** often means that you forgot to initialize (set a default value) in a data structure.
- An ***Out of Bounds error*** usually means that you tried to access a specific position within a data structure, but that position doesn't exist

## Lightweight Inspection Using Print

Print statements can help you narrow down a problem. Let's take a look at an example where an `addLast` method you defined is not passing the tests you wrote :

```
01|  /*
02|  * An example test that you are failing
03|  */
04|  @Test
05|  public void testAddLast(){
06|    MutableList<Integer> myMutableList = new MutableList<Integer>();
07|    myMutableList.addLast(4);
08|    myMutableList.addLast(3);
09|    myMutableList.addLast(2);
10|
11|    assertEquals(4, myMutableList.getFirst()) //test that you are failing
12|
13|    System.out.println(myMutableList); // 2, 3, 4
14|  }
```

The above code adds three integers by calling `addLast` three times. If everything is correct, then our `MutableList` should be `[4, 3, 2]`.

However, the `System.out.println()` at the end of the method prints `2, 3, 4` which is in the reverse order! Now you know why your `addLast` method is not working - it's not adding the node to the end of your list.

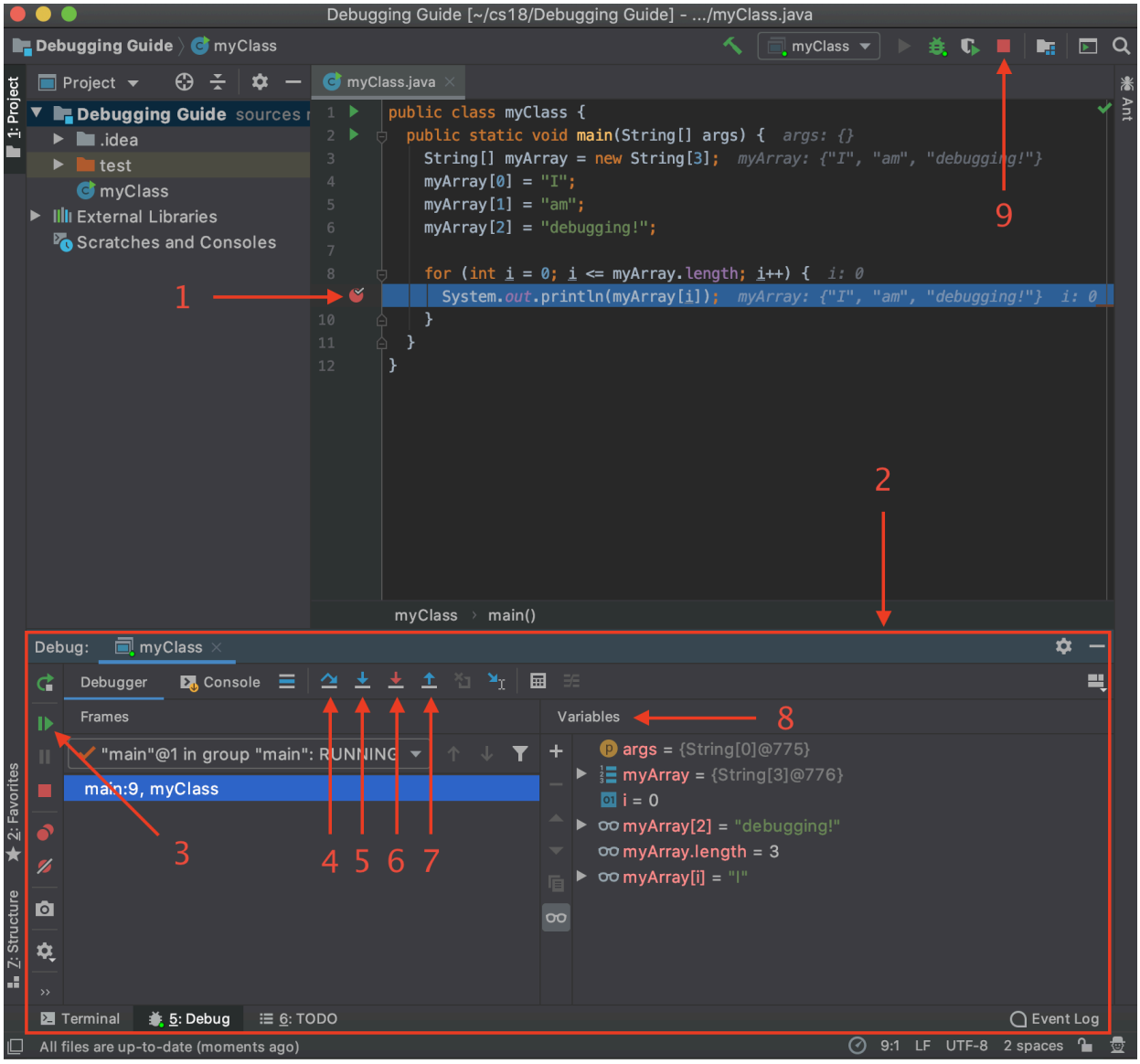
Print-based debugging can be effective for showing what's happening at a high level. Sometimes, knowing the high-level problem doesn't help us find the low-level problem. That's where we turn to the debugger for deeper help.

## Intellij's Debugger (covered in lab 3)

To start debugging, find the bug icon in the top right corner. This button runs the program indicated by the dropdown menu to the left of it.


Before running the debugger you have to set up **breakpoints**, stopping points where you can observe the state of your program. The console gives error messages indicating where the program failed, which often serve as good places to set breakpoints.

After setting a breakpoint and running the debugger, Intellij provides you with a debugging window, giving you access to all of its features.




1. **Breakpoint** - When running the debugger, your program pauses each time it encounters a breakpoint. Note that your program pauses *before* executing the line it highlights.
2. **Debugging Window** - This is where you will see everything you'll need to debug your code such as all the useful debugging buttons and variables related to your program.
3. **Resume** - The resume function continues running your program until it reaches another breakpoint or terminates.
    a. This function is helpful (among other uses) for cases when you are using recursion or a loop and want to see how the state of your program updates for each recursive call or iteration of the loop.
4. **Step Over** - The step over function does not enter a function but instead jumps over to the next line of code.

    a. This function is really useful for seeing how variables change as you run your code and allow you to go through a method line by line.
5. **Step Into** - If a function is encountered or you had a breakpoint at a function call, step into will stop program execution at each line in the function and show you how each line gets processed
    a. This function is helpful if you are stepping through a method (see **step over** above) and encounter another method call and want to see how the new method call affects the state of the program.
6. **Force Step Into** - Won't be needing or using this action but it lets you debug methods defined in the APIs or Libraries.
7. **Step Out** - This will let you skip stepping through code in a method line by line and return to the calling method
8. **Variables Panel** - This is where you will see all the variables you have defined in the method you are debugging. By clicking on each variable you can see the values associated with them which can help you determine whether your variables are storing the values you expected them to store. Stepping into your code and monitoring your variables to make sure they are storing reasonable values is what will help you locate your issue as quickly as possible!
9. **Stop Button** - If you want to stop debugging mode, you can click on the red square stop button in the top right corner next to the debugging button or on the left hand column of the debugging window.

## General Tips for debugging

- Create informative variable names.
- Set breakpoints often.
- Don't step into code you didn't write.
- Take breaks! Sometimes all it takes is a clear mind to solve the issue.

---

*Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS200 document by filling out the anonymous feedback form!*