The CS18 Guide to Debugging

Intro

As you will come to see, bugs are, tragically, a staple of CS. Regardless of how experienced you are, it is almost certain that you will come across bugs in your code that will require a significant amount of time and energy to find. This is especially true in a class like CS18, where there are projects with large numbers of moving parts, and where we allow mutation of objects. With this guide, we aim to help you get better at debugging. Additionally, we have provided a few examples of bugs especially common to CS18 that you might come across with the hope that you can use this guide to solve them quickly.

Why is the Eclipse Debugger useful?

While putting printlines in your code to check for intermediate results is a valid strategy, we ***HIGHLY*** encourage that you use the Eclipse debugger instead. This debugger offers a number of benefits over printlines, including the following:

- Giving you access to the whole namespace - When you make printlines, you often only print the value of one/two variables. However, it might actually be that some variable which you didn't print is really the source of the problem. With the debugger, you see the value of **every** variable in the namespace/object, so you can see exactly what is going on.
- Letting you move step by step - The debugger gives you the option to step through your code line by line, seeing exactly how everything changes at each step. This level of fine grain detail is not easily captured by printlines, and it is far easier to track the change of a value across the lines of your code with this.
- Letting you debug even with complicated structure - In 18, as the semester progresses, you will begin to code things which don't necessarily lend themselves to printlines well. Consider a class with 5 different fields. How will you use a printline to check all of those fields? It is likely that you will have create a toString method, which then prints out the

field values, though if these fields are also complicated classes, then you need toStrings for them. With large systems, it is impractical to use printlines, but with the debugger, the classes are defined right in the namespace, and you are able to look exactly at what happens in even the most complex classes at every step of your code.

Eclipse Debugger Instructions

How to access

To get set up, click the bug button at the top right of the Eclipse screen. This will arrange your windows in a way that is more suitable for debugging.

- The first time you do this, a popup might appear asking about switching perspectives. You should go ahead and approve it!

If the bug button is not visible, you can switch to debugger mode in eclipse by clicking Window > Open Perspective > Debug on your menu toolbar.

Debugging Tools

- Breakpoints:

  One of the most useful things about a debugger is the ability to allow you to set breakpoints. Setting a breakpoint on a line of code, then running your program, will pause your program each time it reaches that line. It will pause *before* executing the line. At that time, you can investigate the state of your program by looking at the menu of variables in the Variables pane located in the top right quadrant of your debugging view.

  To insert a breakpoint, move your mouse to the left of the line you want to break on. If you have line numbers enabled, it should be to the left of the line numbers. Then, double-click, and a blue dot should appear. To remove it, right-click it and select Toggle Breakpoint, or double-click on the breakpoint.

- Step Into:

  Once you're stopped at a breakpoint, there are a few options on how you want to continue. Suppose you have a breakpoint on a line that has a call to a method/function. Step Into allows you to enter the method/function that you wrote and pause execution on the *first line* inside that method before it is executed. **Note:** If you Step Into a method that you did not write (i.e. a systems provided method such as System.out.println), you'll reach all kinds of weird places! Not to fret, there are ways that you can get out of this situation.

  To step into, use the left-most yellow arrow at the top of the screen, next to the red stop button.

- Step Over:

Similar to step into, step over is a way to proceed after being paused at a breakpoint. However, whereas step into would pause on the first line inside the method call, step over will execute the method, then break on the following line of code. In effect, step over will execute the current line, and pause on the next line of code, regardless of whether the current line was a method or not.

To step over, use the next yellow arrow, just to the right of the step into button.

- Step Return:

  What if you pressed step into, but you meant to use step over instead, and now you're stuck who-knows-where in some source code from a system call? Not to fear! You can use step return to finish executing the function you're inside, then pause.

  To step return, use the next yellow arrow, just to the right of the step over button.

- Resume:

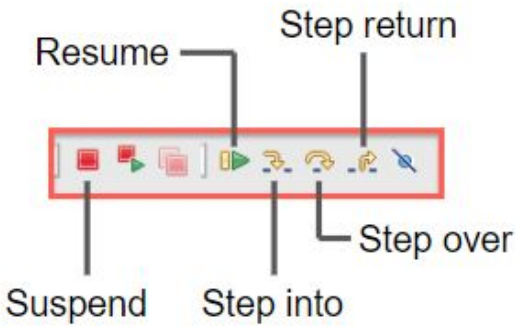
  Yet another way to continue from a breakpoint. This will, in effect, resume execution normally until the next breakpoint or until the program terminates.

  To resume, press the yellow-and-green play button, to the left of the stop button.

- Terminate:

  If you ever find yourself stuck in an infinite loop or you just want to stop your debugger as it is running through the code, use the terminate button.

  To terminate, press the red square button on the toolbar.



To return to the the **normal perspective,** click the multicolored button with a J (for Java) or an S (for Scala) on it in the top right corner.

Feel free to close windows you don't think will be helpful, as you can reset the windows at any time with Window > Perspective > Reset Perspective.

A Step-by-step Guide to Debugging

While the debugger is a fantastic tool, one of the most important skills you can develop is knowing how to locate a bug, with or without the debugger. To some extent, the debugger is a tool for fine grain exploration - it is best used when you can determine some smaller area in your code where the bug is likely stemming from. In this section, we will discuss some strategies that can help you to at least be able to determine a smaller area in your code where the bug might be located. Here are some steps that you can go through to help you fix a bug:

- Start by identifying a concrete input on which to debug your program (you should have this from the program failing). Work through how you expect the program to process this input independently from looking at the code. Independency is essential, because debugging is the act of figuring out how your actual code fails to meet expectations. *You are unlikely to find the bug by any means if you can't get through this conceptual step.*
- Now you have to start isolating where in your code might be causing the problem. If running the code on your input yielded an error message (often in the form of an exception), make sure you understand the issue that Java is reporting and where it gets reported in the code. If instead your code simply produced a different answer than you expected, go to the next step to start isolating the problem.
- If your code modifies data as it computes, use your example to identify the key pieces of data that track the computation internally. Insert some printlns at key places (the start of a loop, the start of a method, etc) to show those contents, then run your program to watch how your data structure evolves. Does it match what you expected from working the example by hand? If not, where does it first diverge? *Note: don't println every variable -- that's what the debugger is for. The printlns are useful for narrowing down the problem and for giving more concise displays of data structures than what the debugger will show.*
- Start the debugger. Put breakpoints at the starts of methods, start of loops, or start of key blocks within your code. Jump to the last point where your printlns produced the expected answer, then step the code to figure out where the code diverges from what you expected to happen. If you aren't able to localize the problem this way, this might be a good time to see the TAs.
- This is where debugging can get tricky. Knowing where a problem appears doesn't tell you why it happens. The error could be anything from a simple typo to a flaw in your overall algorithm in the first place. If you haven't spotted the error yet, try:

    - Checking the information from the debugger against any invariants you know about the behavior of the algorithm (such as what should be true at the start of each method or loop).
    - Using a second example that might take a different path through the code than the first one you started with.

    - Walk through the code again and explain out loud (just to the walls) what is happening. Following your logic, and saying it, can often help you identify where something is going wrong. This technique is so useful in practice that it has a name: rubber duck debugging (because a rubber duck is more interesting to talk to than a wall).

- Remember that additional test cases are your allies. Having test cases that check different parts of your code may help you immensely in narrowing down the problem, and you can run them much faster than you can run the debugger.

Common Exceptions and Errors

For your own use, we've compiled a list of some of the most common Exceptions stemming from various bugs you might run into in CS18. We also have a list of some common ways in which bugs might appear. Note that this list is far from complete, as the world of bugs is large and unforgiving.

NullPointerException

A NullPointerException (NPE) means that the variable/object you are trying to do something with is null (i.e. it does not exist or is not declared).

Common causes:

- Trying to call a method on or in some way use an unset variable. For example, you might have a name field in a class, and by mistake, you never set it in your constructor. When you go to use the name field (perhaps in a printline), you will get an NPE because the variable exists in scope, but is unset, so it defaults to null.
- Trying to access a method/field of a null object. For example, if you were traversing a binary tree, and you come across a node which has no left subtree (i.e., the left subtree is null), you might try to call a method on that left subtree, which would result in an NPE.
- Related to the previous cause, you might also encounter an NPE when trying to test if something is null. For example, you might know that at some point in your code, you will encounter a null value. Inadvertently, you might have your if statements set up in such a way that you test for some things (e.g., test if node.left == null) before you test if the thing itself is null (node == null).

ArrayIndexOutOfBoundsException

This exception arises when you try to access an index in an array that is greater than or equal to the size of the array, or when the index is negative.

Common causes:

- Off by one errors - By far the most common way to get this exception is to miscount just by one. For example, you might have an array of size 10, and accidentally have a for loop that iterates from 0 up to and including 10. This would result in an exception on the last pass of your loop.
- Calculation errors - In a similar vein to before, you might accidentally calculate an index incorrectly, causing it to be negative or too large. This is especially common when you are working with problems where you have to access certain array entries, like if you had to get all the diagonal entries of a 2D array.
- Rounding errors - This is especially true of the base cases of methods, but often you might be asked to find the midpoint of an array, which involves doing some division. Since you are working with ints, Java will round, and might round in a way that causes this exception.

Errors Related to Input/Output (I/O)

When you write programs that involve I/O (i.e. a user providing input), you open up your program to a whole host of weird bugs. Users can sometimes be malicious or inept, and pass input to your system that is so strange that bugs appear. Here are some ways this might happen:

- You read a file which has a blank line in the middle of the file. This might exhibit itself as an NPE
- You read a file which is not formatted exactly to the specification required for your code (e.g, a user accidentally typed a period where there should have been a comma).
- The user passes input to your system that is just the Enter key (i.e, the input is just a return carriage). This may also present as an NPE.
- The user hits the CTRL-D keys, which sends an end-of-file (EOF), when prompted for input. Typically, this is a flag to quit the current program, and in CS18, this is ALWAYS a quit flag. Your programs should handle this gracefully, but if left unhandled, this typically appears as an NPE.
- The user passes ‘invalid’ input. For example, your program may request that the user enter a number, and the user enters a letter. These errors can present as any number of exceptions (like a NumberFormatException), or if handled really improperly, might not throw exceptions at all.
- Related to the previous point, the user might enter an ‘invalid’ input, but because of how your if statements are formatted, this invalid is captured by an else. Concretely, you might have a program that prompts the user to enter a 1 or 2. However, if you write your if statement checking if the input is 1, do something, and else, do something different, you can run into bugs. Instead, you should concretely match on each input, then have an else be reserved to catch this invalid input.

Other General Bug Sources

Here are a few common coding mistakes that might cause bugs to appear in your code, beyond the mistakes presented above:

- Not handling mutation carefully - One of the great features of OO programming is mutation, but this can be a double edged sword at times. Perhaps, at some point in your code, you inadvertently mutate something (like removing a value from a list) without realizing it. Not carefully handling mutation can lead to strange bugs.
- Shared references - Two classes might reference a shared data structure (e.g. hash table), and this might not be desired behavior, or maybe it is, but modifications that are being performed are not done in consideration of the fact that it is shared.
- Not being careful with when you use readLine - Whenever you are taking input, and you call readLine, you have fundamentally altered the underlying Reader. Be sure that when you do call readLine, you are storing your result as needed, as well as checking to see that you haven't hit the null line.
- Similarly, you need to be careful with Iterators. When you call Iterator.next, you have fundamentally altered the Iterator. If you have called Iterator.next with the intention of doing multiple things with the result, you need to make sure that you assign a variable to the result of Iterator.next, otherwise, each time you call Iterator.next, you will be working with a different value than expected (and potentially might even reach a null that breaks your code).
- Incorrect boolean expressions - Whenever you encounter an infinite loop or some if/else decisions, the first thing you should look at is whether or not your boolean expression is evaluating to the intended value.
- Improperly reassigning values - You might be tasked with swapping the left and right subtrees of a node. Be sure that you are using a temp variable to do this right, e.g,:
    - temp = node.left
    - node.left = node.right
    - node.right = temp

  Rather than
    - node.left = node.right
    - node.right = node.left

  The latter would end up assigning both left and right to the original node.right!