Debugging

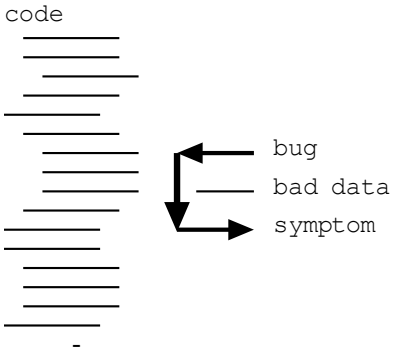
Handout written by Nick Parlante

Attitude

- Mindset
  - It appears hopeless, but there is a logical structure in there. The evidence will be obscure, but consistent in pointing to the guilty code.
  - Avoid "deer in the headlights" -- debugging is the state of mind that although it **appears** impossible, you can sift through it.
  - Don't panic -- be methodical. Somehow the TA is able to do this, and they don't know more about your code than you do.
- Symptom -> Code
  - You observe a symptom -- bad output, an exception.
  - You track from the symptom backwards through the code path to the bug.

Symptom back to bug

- The bug is in the code at point.
- The bug causes bad data, wrong function calls, etc. to happen going forwards, eventually causing symptom.
- The debugging task, in essence, is a backtracking task, starting from the symptom point and working back upstream to find the causing bug.



Great questions

- These questions will solve most bugs:
- What method shows the symptom? What lines of code produces that symptom?
- What is the state of the receiver object in that code? What were the param values passed in? (a breakpoint is a quick way of seeing all those values)
- If it's an exception, what does the exception error message say -- null pointer? array access? Sometimes the exception name can be very informative.

Eclipse debugger

- On an exception -- use the debugger to look at the object ivars and method variables and parameters (the questions above) -- just use "Debug" instead of "Run" in Eclipse to start your program all the time.
- The debugger is good at letting you look at things without having to put in printlns or anything.
- To see state at a particular line, put in a breakpoint (double-click at the left)

Println()

- Println is another way to see state (ivars and params)
- Println is especially good if you want to see the state 20 times in succession in a loop -- the debugger is not as good at that
- For a complex object, take a minute to override toString() so you can just print the whole object whenever you want.

Debugging Mechanics...

1. What does the exception say?

- Exception printouts can look a bit cryptic, but there's often actual info in them -- null pointer vs. array out of bounds. Usually, the exception will list the file and line number where the exception occurred.

2. What Method and line-number ?

- What method execution produced the symptom?
- What line in that method?
- **Map the symptom you observe to a line of code to look at**
- Look at that source code -- half the time the problem is right there. (in emacs, use esc-x goto-line)

3. What is the state of the receiver, parameters?

- The symptom is typically related to bad data -- either in the receiver or in the parameters passed in.
- What is the receiver state? What are the parameter values?
- In the debugger, you can just look right at the ivars and parameters
- Print receiver state -- println(this) will use the toString() of the receiver.
  - Look at all the ivars at the time of the exception.
- Before
  - What about the state a little upstream -- what was the state of the receiver before this message? When did it go bad?
- How did that happen?
  - How did the ivar get that way -- what state did the constructor init it to? What methods changes that ivar? Put breakpoints or printlns in to see the state change over time (especially easy if all changes to that ivar go through a single setter method).
  - Just search for that ivar in the source -- it can only be changed at a line where it appears on the left of an "=" -- just search for those and think about each one.

4. Comment Out / Mess With Code

- Suppose you know the state of the receiver is bad, and you are trying to figure out which code is messing it up. Commenting out calls to sections of code is a very fast way to eliminate code from suspicion.
  - e.g. Suppose you have a draw program and the shapes are in a bad state. Is the move code or the resize code? Comment out the resize and try it -- the program is barely functional, but it's a quick way to decide that code is not the source of the problem.
- Be willing to dork your code around into absurd states to test a hypothesis.
  - e.g. Suppose you suspect the bug has to do with a case where there are many foo objects with a width over 500. The standard code reads the foo objects out of a file. To test the hypothesis, put a foo.width += 500 statement in the file reading code. This line is not very logical for the proper functioning of the program, but it's a very fast way to test the hypothesis.

10 Truths of Debugging

This is a list I built way back when for use in Pascal … but the basic truths still work today.

- Intuition and hunches are great— you just have to test them out. When a hunch and a fact collide, the fact wins. The values of the variables are never wrong.
- Don't look for complex explanations. Even the simplest omission or typo can lead to very weird behavior. Everyone is capable producing extremely simple and obvious errors from time to time. Look at code critically— don't just sweep your eye over that series of simple statements assuming that they are too simple to be wrong.
- The clue to what is wrong in your code is in the values of your variables. Try to see what the facts are pointing to. Be systematic and persistent. The bug is not moving around in your code, trying to trick or evade you. It is just sitting in one place, doing the wrong thing in the same way every time. A line is setting a variable incorrectly, and that bad data is flowing forward to produce the symptom you observe. Sprinkle a few tests of that variable widely to home in on where it goes bad.
- If your code was working a minute ago, but now it doesn't— what was the last thing you changed? Note: this incredibly reliable rule of thumb is the reason your section leader told you to test your code as you go rather than all at once. If you run your program each time you've added 50 more lines of code, then you'll almost always know which 50 lines are responsible as soon as things begin to not work.
- Do not change your code haphazardly trying to track down a bug. This is sort of like a scientist who changes more than one variable at a time. It makes the observed much more difficult to interpret, and you tend to introduce new bugs.
- If you find some wrong code which does not seem to be related to the bug you were tracking, fix the wrong code you found anyway. Many times the wrong code was related to or obscuring the bug in a way you had not imagined.
- If you have a bug but can't pinpoint it, then you should be able to give an argument to a critical third party detailing why each one of your functions cannot contain the bug. One of these arguments will contain a flaw since one of your functions does in fact contain a bug. Trying to construct the arguments may help you to see the flaw.
- Be critical of your beliefs about your code. It's almost impossible to see a bug in a piece of code when your instinct is that the code is innocent. You will sweep your eye over the code 20 times, and never see the problem. Only when the evidence has narrowed things without question to that piece of code, will you suddenly be able to see the bug.
- You need to be systematic, but there is still an enormous amount of room for beliefs, hunches, guesses, etc…. Use your intuition about where the bug probably is to direct the order that you check things in your systematic search. Check the functions you suspect the most first. Good instincts will come with experience.
- Debugging depends on an objective and reasoned approach. It depends on overall perspective and understanding of the workings of your code. Debugging code is more mentally demanding than writing code. The longer you try to track down a bug without success, the less perspective you tend to have. Realize when you have lost the perspective on your code to debug. Take a break. Get some sleep. You cannot debug when you are not seeing things clearly. Many times a programmer can spend hours late at night hunting for a bug only to finally give up at 4:00 a.m. The next day, they find the bug in 10 minutes. What allowed them to find the bug the next day so quickly? Maybe they just needed some sleep and time for perspective. Or maybe their subconscious figured it out while they were asleep. In any case, the "go do something else for a while, come back, and find the bug immediately" scenario happens too often to be an accident.