Java Tools and Debugging

Java On Unix

See the "java on unix" page in the java tools section off the course page. Everyone will need to be able to compile and run on unix.

.cshrc setup

"which java" should yield /usr/pubsw/bin/java

"echo $CLASSPATH" should either be undefined, or will list directories and .jar files that should include the current directory "."

Compile and Run on Unix

"javac *.java" -- compile all files in directory

"java MyClass arg1 arg2" -- run main() function in class MyClass (searches current directory if CLASSPATH is empty or includes ".")

Note that GUI programs will throw a bunch of "font not found" warnings -- you can ignore those. The program will, in fact, work fine.

CodeWarrior

Stanford has a site license for Codewarrior 7 Mac and PC. It should work on MacOS X also. We may get a Linux version someday.

You should have some familiarity with the Unix tools, but using a real IDE is quite handy.

/usr/class/cs108/bin -- utilities

Little unix utilities for common problems -- finding java sources, fixing EOLNs in files, removing tilde files, running VNC.... see the README in the directory.

You can add the directory /usr/class/cs108/bin to your PATH in your .cshrc file to access these utilities. (See the Java on Unix page for instructions)

Java Web Docs

http://java.sun.com/j2se/1.3/docs/api/index.html (linked from course page)

Click on the lower-left pane, and use your browser's search feature

Open things in new windows to avoid reloading

Java Sources

Note that the java sources are the property of Sun -- looking at them may limit your ability to write a non-Sun "clean room" re-implementation of the java libraries in the future.

The java sources contain the HTML text that was used to make the web docs above.

Sometimes looking at the code itself gives insight -- e.g. AbstractCollection.toString()

Debugging

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. You cannot take on a passive attitude.
- 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.
- What method shows the symptom?
- What is the state of its receiver?
- What were the param values?

debugger vs. System.out.println()

- Java debuggers are of uneven quality. Many programmers use System.out.println() for everything.

Test your assumptions

- If all the evidence points to the foo() method being wrong, but you are positive that foo() is right, go look at foo() again anyway. Put in some more printlns. Test your assumption.

Get Some Sleep

- If your brain is worn out, you can't debug well. You become too fixed in your assumptions to see what the evidence is trying to tell you. Often times the next day, you will see the bug in 5 minutes.

Debugging Mechanics...

1. What does the exception say?

Cryptic

- Exception printouts look a bit cryptic, but there's often actual info in them -- null pointer vs. array out of bounds.

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)
No line numbers

Usually, line numbers work even if you don't do anything special
If there are no line numbers, try compiling with javac -g *.java
Also, you can run without the runtime compiler with
"java -Xint MyClass"
Or run under jdb -- it prints the line number and exception (See the Java on Unix page for jdb info)

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.
Parameter values -- are they good?
Print receiver state -- toString
Look at all the ivars at the time of the exception.
Its handy to have a toString() utility that just dumps the state of an object to a String for debugging
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).

4. Comment Code Out / Mess With The 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.