CSCI241, Fall 2018, Lab 3
Due Tuesday, January 29th at 9:59pm

Unlike the homework assignments for CSCI241, you are **encouraged** to work with your peers in completing the labs. However, each student must have their own submission; you cannot exchange files. If any of this is unclear, please ask for further clarification.

1 Overview

In Lab 2 we discussed testing. Once tests have detected a problem, you need to locate and fix the problem. This is known as *debugging*. Every programmer deals with bugs. As one gains experience, it becomes easier to *find*, *fix*, and *avoid them in the first place*, but they never entirely go away.

Introductory computer science students, however, tend to have an especially difficult time debugging, which can interfere with both learning and the inherent fun of programming. However, the same bug that stalled a student for hours can often be found and fixed within minutes with the help of the instructor. Part of this can be attributed to a fresh pair of eyes, but the primary difference is the instructor's understanding of the art of debugging.

This lab is designed to convey some of the fundamental strategies in debugging using a command line debugger called `jdb`, for java Debugger. Most other programming languages have similar command line debuggers, including `gdb` for gnu debugger. The goal of this lab is for you to improve your ability to effectively handle bugs.

There are several kinds of software bugs, but they all have one thing in common: they cause your program to behave in incorrect or unintended ways. Whereas some syntax bugs can be caught by the compiler (all of those compiler errors you love), others cannot be detected automatically. This lab will focus on *logic errors*; that is, when your program dutifully executes the instructions you gave it, but unfortunately your instructions do not produce the intended outcome. Logical bugs are often the most difficult bugs to find and fix.

2 Git and submission for Lab 3

The Github Classroom link for Lab 3 is available in the Lab 3 assignment on Canvas. Clone your repository as you did for Lab 1; be sure to clone it somewhere outside your other lab and assignment repositories to avoid having nested local working copies of different repos. As usual, you will submit your code for this lab by committing and pushing the completed files to your remote Lab 3 repository on Github before the deadline.

It is recommended that you `git add` *and* `git commit` *regularly while developing your code; e.g., one or more times per method you implement. When you have something working,* `git push` *your changes to GitHub.*

Whether or not you finish the entire lab during lab period, make sure to commit and push what you have at the end of the lab period to receive credit for attending lab.

3 So you have a logic bug...

The good news is that most of what you have written is correct (probably). The bad news is that at least one thing you have written is not correct (otherwise your program would produce the intended output), and now you have to find it. Inevitably, there will be any number of places where the bug could be.

Stop for a second, and take a deep breath, because this is where many students go horribly wrong. Do not start randomly changing your code, hoping that the next little tweak will solve your problem. Frantic, disorganized debugging is not likely to solve your problem, and will often worsen it. You may also want to add and commit your code in your repository before debugging, so you have a clear record of all debugging changes you made.

Then, carefully and systematically isolate the problem. Whereas when you begin, the problem is "in your program" or "in this class," as you continue to drill down your problem will become "in this particular function" and then, for example, "in this loop." This can be done with increasingly specific tests. As you test smaller and smaller components, you can deem certain parts of your code as more or less likely to be the culprit. Sections 4, 5, and 6 of this lab provide three strategies for localizing your logic error.

4 Comment things out

This one is particularly handy for detecting runtime exceptions. If you comment out a particular chunk of code, and the problem still persists, it reduces the odds that your bug is in that chunk. Likewise, if after commenting something out the problem goes away, you're probably on the right track.

5 Print statements

Using print statements to inspect the value of variables during the running of your program is a simple – and sometimes, but not always – and effective way of debugging. However, if your program is complex and requires the use of many print statements to debug, the the amount of information printed to the screen becomes unmanageable.

What you don't know about your code can certainly hurt you. When there is a bug in your code, often the state of the program (the memory used by the program, including the variables) differs from what you would expect. You can use print statements to keep a closer eye on the state of the program in two key ways:

1. Print the value of variables at key points in your program. When you run your program, watch to see if any of the variables are being set incorrectly.

2. Use print statements to serve as signposts for where you are in the code (e.g. stating that you've entered in to a function, loop, if statement, etc). For example, you might have the program print "inside of the for loop, and i has the value 4." This lets you monitor the flow of the program as it runs; watch for any suspicious deviations from the flow you expect.

6 The java debugger tool

A good debugger can be used (often more effectively) to serve the same purposes as print statements: to let you see the state and execution flow of your program. Some debuggers even offer functionality that lets you unit test on the fly, or simulate commenting code out. The next section focuses on one particular debugger for Java: jdb. Note that this lab is introducing you to the command line debugger. Much of the same functionality in jdb is available in IDEs, which permit you to set break points, and inspect the value(s) of a variable when a program is running. If you normally program in an IDE, you are encouraged to learn how to perform the same debugging functions in the IDE; however, it's good to learn how to use a command line debugger first.

Lab Description (debugging with jdb)

The Java Debugger, jdb, is a tool to help you find bugs in your Java programs. Below is a step-by-step walk-through of key jdb functionality.

1. As evidence that you used the debugger, this lab asks you to record your command-line commands that you issue. That is done by using the `script` command line program, which allows you to record your keystrokes as well as output generated by a program.

   Start logging with script using the following command and leave it recording for the remainder of the lab. When you are done, you need to explicitly end the script, otherwise the script file will be blank. You will turn in the transcript of your commands.:

   ```
   script -a --timing=typescript.timing
   ```

   Note that the `-a` means append, so if you work on this lab in multiple sessions (on the same machine, or on lab machines where your files persist across logins on all computers) it will simply add your new commands to the previous session's command log. Call this command at the beginning of each session, and you'll have a complete record of your work in a single typescript file. The `--timing` argument means you will log the timing of your session, which makes it easier for the TA to see the session as you saw it.

2. You'll find the file *SelectSort.java* in your lab3 repo. It is a file that contains at least two logical bugs. Compile your code with debugging support:

   ```
   javac -g SelectSort.java
   ```

   The "-g" option tells the compiler to generate extra information useful for debugging.

3. Start the debugger (`jdb`):

   ```
   jdb SelectSort
   ```

This line starts the debugger with the program symbols (debug information) of your compiled code (`SelectSort.class`).

4. Check out the available options by typing `help` at the prompt `jdb` gives you. This will output a long list of commands that will be very helpful in learning to use jdb. Feel free to type this command anytime in jdb when you cannot remember a command, or want to learn a new command.

5. To run your code inside the debugger, type `run` in the prompt. If your program takes commandline arguments, you can pass them as follows:

   ```
   run <class> <args>
   ```

6. However, you often want to control the pace at which you proceed through running your program, so you can spot where things first go wrong. One way to do this is to step through the program line-by-line. You can do this with the `step` command. When run, it will execute the current line and stop on the next.

7. Sometimes you have a good idea where the bug is, and it might take a prohibitively long time to step line-by-line to get to the area of interest. In these cases, you can you typically want to set *breakpoints*, which tell the debugger where to stop running the code and return control to you. You can set a breakpoint at a particular method with

   ```
   stop in <class>.<method>
   ```

   In that case it will run until this method is called. You can also set a breakpoint at a particular line of code with:

   ```
   stop at <class>:<line>
   ```

   With breakpoints, you don't need to step from the get-go, you can `run` your program and it will return control to you at the first breakpoint it encounters.

8. If you want to resume running your program after you have stopped at a breakpoint, you can type `cont` (continue).

9. To list all current breakpoints, type `clear`.

10. To remove a breakpoint, type

    ```
    clear <class>.<method>
    ```

    or

    ```
    clear <class>:<line>
    ```

11. Other assorted useful jdb commands are listed below. Pick at least three of them and try them out in the debugger:

    - You can add a class variable (aka field) to the "watch list" so that you see its value each time it is updated:

      `watch [access|all] <class>.<field_name>`

      Note: You cannot put a "watch" on local variables
    - `next` acts like `step` but it doesn't enter called functions
    - `print <var>` prints the value of a variable
    - `dump <var>` prints object details for an object variable
    - `locals` lists the local variables
    - `where` dumps the stack of the current thread
    - `list` shows you where you are in the code
    - `!!` repeats the last command
    - *Don't forget to check out* `help` *for many other useful commands...*

12. There are (at least) four bugs in `SelectSort.java`. Use the strategies introduced in this lab to find and fix these bugs. Hint: you may also find it useful to write additional tests to help narrow down where the bug might be.

Rubric, and what to hand in

Make sure you have committed your fixed SelectSort.java file. Be sure to push your committed local changes to github. As always, **do NOT push .class files to your git**.

Your lab3 repository should contain the following files:

- `typescript`
- `typescript.timing`
- `SelectSort.java` (with all bugs fixed)

Each bug fix is worth two points. Completing all jdb walk-through steps is worth two points. Deductions will be made for the following:

- Missing or incorrectly named files
- Introducing poor coding style (e.g., inconsistent indentation, bad variable names, etc) into `SelectSort.java`
- Other assorted failure to follow Lab 3 instructions

Acknowledgments

Many thanks to Jonny Mooneyham, Tanzima Islam, Qiang Hao, Brian Hutchinson, and Filip Jagodzinski for producing and refining past labs on which this lab is based.