CS 455 Lab 6: Using a Debugger

Fall 2020 [Bono]

Goals and Background

This lab introduces you to a symbolic debugger to use with Java programs. The debugger is part of the free Eclipse IDE. Eclipse is not installed on the campus lab machines. If you haven't already done so you will need to install Eclipse on your own laptop ahead of time and then bring your laptop to lab with you.

Even if you figure out some of the errors in this lab without the debugger, you should go through the practice of using the debugger to get more information about your program; debuggers are useful tools for every programmer, this is a good opportunity to acquaint yourself with one. For some reason students aren't motivated to learn a new piece of software like this when they have a mysterious bug in a program due in 3 hours :-) so we'll give you points for trying it out now!

Note: if you are already familiar with another full-featured IDE you can use that instead of Eclipse, but the directions for how to do things here are only about Eclipse. If you hadn't already used the debugger in that IDE, you would need to figure out how to do actions such as setting a breakpoint in that debugger before the lab.

Reading and reference material

- Horstmann, Section 6.10 Using a debugger
- Eclipse Environment Setup (pdf) (Also linked from the Documentation page.)
- Java Compiler Tutorial. From Cay Horstmann's web pages. In your browser, search for "Eclipse" to get to the section of the document about using that tool. (Also linked from the Documentation page.)

Exercise 1 (1 checkoff point)

1. Download the starter files for the Lab 6 item from Vocareum onto your laptop or the lab computer. In the Vocareum IDE you do this by clicking on the Actions menu on the upper right, and select "Download startercode." The files are:
   - `Factors.java`
   - `NamesTester.java`
   - `Names.java`

   Unless you already have an eclipse workspace where you normally keep your code, you can make a local `lab6` folder to be the workspace, and then two subfolders, `factors` and `names`. Put the source files related to those two programs in the appropriate subfolder.

   When you open Eclipse you can tell it to use the `lab6` folder you just created as the workspace.

2. `Factors.java` is a program for calculating all of the factors of a number. Make a project for the Factors program in Eclipse (directions follow here for those that haven't done this before):
    - Choose the menu option File > New Java Project.
    - Choose `factors` as the name of the project, and it will use the `factors` folder and the code inside it for the project.

3. The program does not quite work as is. Compile the program; then run it, responding to its prompts.

    This is probably not the first time you have seen the results of a program-crash in Java. Both when using `java` from the command line, or running in Eclipse, the output that you get when a program crashes is a descriptive exception name, which may clue you in to why the program failed. Below the error message the subsequent lines of output show the call stack, that is, a list of the methods that were called, but not yet returned from, when the program failed. If you look from the bottom to the top of this list, at the bottom you'll see your `main` method, then above that the method that was called by main, and then the method that was called by that one, etc.

    The top line of the call stack output shows exactly where the program failed: the class, method, the source code file, and line number. In Eclipse, if you select that information with your mouse, it will highlight that line of code in its editable source code display window. Do that now.

4. Once you are viewing the part of the code where the crash happened you may be able to figure out what the problem is. Even if you know what the problem is, let's see how we can view more information about the values of variables at the time of the crash:

    How to start the debugger: You are going to run the program again. Instead of choosing Run on the Run menu (or "Run as" from right-clicking the main class in the Package explorer window (left side of screen)), choose Debug on the Run menu (or "Debug as" via a right-click). Enter data for the program as before. Once the program crashes it will ask you whether you want to change to the Debug perspective. Answer yes. This will show a different set of windows, although your console window is still at the bottom.

    How to inspect variables in the debug perspective: Once the program is stopped in the debugger you can see the values of the variables at the point where it stopped. All the variables that are currently in scope are shown with their current values in a window at the upper right of the screen.

    How to look at variables in other methods on the call stack: Sometimes the actual problem is in the method that called this method, or some other method lower in the call stack. (More about this below.) Once the program has crashed or stopped at a breakpoint, you can look at the values of variables in other methods on the call stack as follows: look at the the Debug window at the upper left of the screen; select the desired method in the call stack list that appears there.

    ---

5. Follow the directions in the last paragraph above to look at the variables in `main` too. There isn't much of interest there, but this is useful for more complicated programs. If you want to go back to the variables in the `factors` method, just select it on the call stack listing.

6. Fix the error that's causing the program to crash. The error can be corrected by changing only one line. (Note: there will still be another bug in the code; we'll work on that in Exercise 2.)

More about run-time errors: (Note: you do not need this information to complete the exercise, but it's useful for debugging in general.) When you get a runtime error the location where it crashed is not necessarily the location of the buggy code. E.g., if you get a null pointer exception while trying to call a method on an object reference, the problem may not be the method call, but that you failed to initialize the object reference properly, and that code may be in a completely different method than where the error occurred. For example:

```
public void foo(MyClass myObj) {
    myObj.myObjMethod();     // <-- null pointer exception here
}
```

Unless foo is supposed to handle null values, the error is somewhere else in the program, likely somewhere in one of the methods that directly or indirectly called foo.

The crash may even happen in Java library code; but that doesn't mean there's a bug in the Java library code. For example, if you go off the end of an ArrayList, the method shown at the top of the call stack is some ArrayList method that's called by `get` to do range-checking, but the actual error is not in `get` but that you violated the precondition of `get` by passing it an out-of-range index.

Exercise 2 (1 checkoff point)

1. Before proceeding, read about a few more commands that will come in handy:

How to get in and out of Debug perspective: There are buttons at the top right of the screen that will switch between Java and Debug perspectives. Alternately, you can use the "Open perspective" option on the Window→perspective menu.

How to abort a program: When you are running or debugging a program you can stop it by clicking on the solid red rectangle icon that's right above the Console window, or by choosing "Terminate" option from the Run menu. (You may already know that for a program run in Linux you can abort by typing `ctrl-c`.)

---

2. Recompile your new version from Ex. 1 and run it to see what it does.

3. It would be useful to be able to stop the program to see exactly what is going on. We are going to set a breakpoint in our program, and then run it so it stops at that breakpoint.

Choose a likely place that would be causing the problem, and set a breakpoint there:

How to set a breakpoint: Select the line you want to put the breakpoint on (or remove the breakpoint from), and from the Run menu select the "Toggle breakpoint" option (alternatively you will see this option if you right click on that line). It will show a symbol to the left of the statement indicating a breakpoint has been set there. To run to your breakpoint, you need to choose Debug (Run will go right past it). The statement where the program stopped (i.e., the one with the breakpoint) will be highlighted in green and have an arrow to the left of it.

How to clear a breakpoint: For when you are still debugging but no longer want to stop at a particular point, you can just do "Toggle breakpoint" again at that statement to turn it off.

Once your program has stopped at a breakpoint, you will be able to see current values of the variables at the upper right like we did before. But since our program is still active this time (unlike when it had crashed), we can keep running the program and see how the variables change. Read about single stepping before proceeding.

How to single step: Your program needs to be stopped at a breakpoint to start single stepping through it. There is a set of icons above the Debug window (upper left) that are debug commands. Hover your mouse over them to see the associated command descriptions. (These commands are also available on the Run menu.)

There are two main kinds of single step commands:

- **Step into** Goes to the next step in the program, even if this means entering a method call. (It stops at the first line of code in the called method.)
- **Step over** Runs to the next statement in the method you are stopped in, without stopping inside any method calls (it will run a method called from here, you will just not have to stop at every statement in the method).

When to use which one? You will step over method calls that you know work, e.g., because you already debugged them, don't suspect a problem there, or because they are Java library methods. You will step into methods that you are still working on, or where you suspect the bug is. If you step into a method by mistake, use **Step return** to skip over the rest of it.

---

4. Run your program until the breakpoint. Once your program is stopped because it reached the breakpoint, single-step to follow the control flow, looking at the values of variables as you go. Think about what's happening in the program, compared to what we want to be happening.

   In trying to diagnose a bug, running until the next breakpoint, instead of single-stepping, can also be a useful technique:

How to resume running the program: The green triangle above the Debug window resumes program execution. It will run until it reaches the next breakpoint, or until it reaches the end of the program. For example if a breakpoint is in a loop, it would run until it hits it again in the next loop iteration (unless the loop exited).

---

5. Fix the error. For check-off, show the TA your working program.

Exercise 3 (1 checkoff point)

We're going to work with NamesTester.java and Names.java for the next two exercises. Make a project in eclipse for this program.

Now we're going to focus on displaying more complex data structures in the debugger. The version of this class and tester we're using for this lab is a little different than the one used in lecture: this version has code to allow the client to insert, remove, and lookup names that are internally stored in order in an `ArrayList`. This is a different representation than we used in our lecture version, but we want to use it here so you can see how to display the internals of a Java object.

Like the version we did in class, here we have a test driver `NamesTester.java` that tests all of the methods on several test cases each. (Note: The NamesTester.java version is also a little different than the one we used in lecture.) In addition to the methods we did in class (lookup, remove), `Names` has an insert function which inserts names in sorted (i.e., alphabetical) order. You may want to examine NamesTester.java to see how the insert method gets tested.

The Names insert method does not quite work correctly; run NamesTester see how it behaves.

Now, run this program again in the debugger to try to figure out what's wrong with it. Set breakpoints to stop execution at useful locations. Hint: so you don't spend a lot of time single-stepping through test cases that might not give us much information, stop the program at the first call to `doInsert` (call is in NamesTester) that gets incorrect results; then you can step into the call to `insert` that has the incorrect behavior.

Once you have a breakpoint set, you can view variables as we did before, but this time some of the the variables are object references and arrays. For these more complicated objects, you can view the subparts by clicking on the + next to the variable name. You may have to do this for a few levels to see everything you want to see (e.g., this --> namesArr --> namesArr elements).

Once your program is stopped in the insert method, make sure you can see the following key data for this method: namesArr individual elements, namesArr size, newName (the one we are trying to insert), and targetLoc. You can make the Variables window larger to see them all at the same time.

Once you get the display set up, you can single-step to see how the values change as you do each step in the program. You may want to examine additional variables as you go as well.

You get this lab point by (1) showing the TA the contents of `namesArr` (including the strings inside it) being displayed in eclipse while your program is stopped at a breakpoint, and (2) based on what you see, telling the TA what the current capacity of the ArrayList is.

For DEN students, take a screenshot of the display showing the contents of `namesArray` as described above. Call the file `ex3Screenshot.png` (or ex3Screenshot.bmp depending on file format).

Exercise 4 (1 checkoff point)

Single-step with the debugger, seeing how the values change to help you diagnose the problem. Fix the bug. (Note: fix the code by making minimal changes -- ArrayList has methods that would allow us to eliminate one or more of the loops, but we'll take this as an opportunity to practice with array manipulation code: do not eliminate any loops in your fix.)

As you can see from this problem, the debugger is just a tool to help us see exactly what our buggy program is doing. It doesn't actually diagnose or fix bugs. For a non-trivial bug like this one, you may need to give it some thought, including doing some pencil and paper work, before figuring out how to correct the problem. As with all programs, you definitely do not want to just start adding random statements until it seems to work.

Checkoff Summary

Show the TA your correctly working `Factors` program and `NamesTester` program. For Exercise 3, see directions in the last paragraph of that section.

Checkoff for DEN students

Upload your `ex3Screenshot` file and the working versions `Factors.java` and `Names.java` into your Lab 6 Vocareum workspace. (Upload button is on the upper left in the Vocareum IDE.)

When you click the `Submit` button, it will be looking for and compiling (for source code) the files
`Factors.java`, `Names.java`, and a file with the prefix `ex3screenshot` (since screenshots are stored in different formats on different platforms).

This lab is adapted from one written by Mike Clancy.