The purpose of this lab is to introduce profiling. We will use the VisualVM profiling tool to analyze the performance of different programs.

0 Materials

1. `debugging/` - This folder contains the class `TroubledLobster`, whose methods have several bugs. Each statement of the main method will encounter one of the bugs in these methods, and each bug will crash the program at some point during the run of `main`. The folder also contains an image which is shown at the end to congratulate the user on having fixed all of the bugs.

2. `profiling/` - This folder contains two classes, one program that does regular string concatenation (indeinitely) and the other that uses a `StringBuilder` object to do the same string concatenation (again, indefinitely.)

1 Eclipse Debugger

1.1 Introduction

Debuggers give you fine-grained control over the execution of a program and allow you to examine both the behavior and flow of the program as well as the contents of any variables in scope. Debuggers allow you to look more closely at areas of code where you believe the behavior may not be exactly what was intended, and verify the control flow and variable contents are what you anticipate them to be. Debuggers provide the functionality to run a program normally up to a specified `breakpoint` in the code, suspending the thread at a place where you can examine the state of the program. It is also possible to step through code one line at a time.

1.2 Setup

No real setup is required for the Eclipse Debugger. The Debug perspective can be switched to explicitly. This view has several useful features: the breakpoints window, the variables window, a console, and run options when debug is enabled (The bug icon or Run → Debug.)
1.3 Useful Debugger Features

1. **Breakpoints** - Breakpoints can be set at certain lines so that the execution of the thread will suspend at this point when run in Debug mode. Breakpoints are usually set slightly before an area of code where you suspect something is going wrong, so that the flow of the program and contents of variables can be examined up to the point you are seeing the error occur.

   Breakpoints can be set either by right clicking on the line number in Eclipse and setting 'Toggle Breakpoint', or by selecting a line and going to Run → Toggle Breakpoint.

2. **Conditional Breakpoints** - Once a breakpoint is set, it can be made conditional by right-clicking the breakpoint in the lefthand margin of the code or in the Breakpoint window, and selecting Breakpoint Properties. You can set the Conditional radio button and specify a conditional for when the breakpoint is or is not to be in effect.

   Sometimes it is useful to add additional code to implement the condition under which a problem is known to occur so you can use it to define a breakpoint.

3. **Watchpoints** - Watchpoints behave similarly to breakpoints, but are set on fields of a class. If a watchpoint is set on a field, the debugger will suspend execution every time the value of the field is read or modified (though the preferences can be modified to only do one/both of these.)

4. **Exception Breakpoints** - Usually when the debugger encounters an exception, it will end execution and show the stack trace, as would happen when you normally run the program. If you would like the run of the program to suspend when an exception is encountered, an Exception Breakpoint can be set to look for any Java Exception. This is useful when you want to examine the contents of memory when an exception occurs.

   An exception breakpoint can be set thorough the Run menu, like the others, or by clicking the icon with a J and an exclamation point in the Breakpoint window.

5. **Step Over** - Step Over will run the current line and advance to the next line, without showing the user any of the code that is executed within that line. For example, if you call a function in this line, the debugger will execute this method as normal and suspend execution at the next line of the function you are currently looking at.

6. **Step Into** - Step Into will run the current line, and jump to the code that is executed by a function call, if any, in that line. If a method is called in this line, the debugger will only run to the first line of this called method before suspending again.

7. **Step Return** - Step Return runs the rest of a method that has been stepped into and returns to the calling method.

8. **Run to Line** - If you don’t want to set a permanent breakpoint, you can select any line and simply select Run to Line. This will not create a breakpoint, but it will act like one temporarily.
1.4 Demo

The demo consists of only once class, TroubledLobster, which has several bugs. The included main method runs some of the Lobster’s methods in order to make some of these bugs manifest themselves. If all goes well, you should be able to use the debugger to place breakpoints in the code and track down all three major bugs - you’ll know you’ve found them all if main() is able to call the last function!

1. Begin by just running the code normally. You should see an exception pop up in translateToLobster. This error has to do with String equality being checked as == instead of .equals(), and thus the recursive method never reaches its base case. This is a good time to try a conditional breakpoint, because if a regular breakpoint is placed in the for-loop, it will take quite a while to get to a point that is relevant to the error you’re seeing.

2. After you fix this error and try to run the code again, you will find an IndexOutOfBoundsException in the method feedLobster. This error comes from the constructor of LobsterList. When stepping through the feedLobster code, this is a good opportunity to use the StepInto function of the debugger to examine what happens in the List constructor.

3. The LobsterList constructor has two problems: first, it does not call super(ArrayList) and thus the default constructor is called, initializing the List to be empty. Add super(food) to the first line of this constructor to fix the problem. The second error (though it does not cause any exceptions) is that the LobsterList checks for negative elements in the array and throws an exception, but then immediately catches it and does nothing. Is there a better way to handle this?

4. When the code is run again, you will encounter an AssertionError in walkLobster(). This is because the code checks to see if the lobster has legs when you try to walk it, and this lobster, even though we initialize it with 5 legs in main, appears to think it has 0 legs. Where might this error be coming from? Placing a breakpoint is more challenging, because unlike the other bugs, this one does not occur in the method that throws the exception.

5. The actual error is in the constructor, where the local variable num_legs is set instead of the class field num_legs. This should become clear from inspecting the variables in Debug view when stepping through the TroubledLobster constructor. This can be fixed by changing num_legs in the last line of the method to this.num_legs. Once you’ve done that, if all goes well...

6. Rock Lobster! (make sure the image is in your source directory for this to work correctly)

2 Profiling with VisualVM

In the profiling folder, two files are included: StringRepeater and BetterStringRepeater. One uses a StringBuilder to concatenate Strings, and the other uses the addition operator.
this part of the lab, you will use VisualVM to study the performance of the code and to see also highlighting the different features.

2.1 Key Features

1. **Heap Dump** - One of the most used features of VisualVM is the Heap Dump, and it can be done by just right-clicking a running process in the left column and selecting Heap Dump. This will show the memory usage, number of classes, number of instances, and other interesting statistics.

2. **Monitor** - The monitor tab will likely be the most interesting for examining the differences in StringRepeater and BetterStringRepeater. Here, you can see speed and Heap Usage in real time.

3. **Sampler/Profiler** - For performance optimization purposes, the Sampler view is absolutely essential. (The “Profiler” tab uses a different approach to collect profiling data, but gives similar results.) The “CPU” button can be pushed to start collecting samples about the attached process. Sampling stops once the “Stop” button is pushed. You can then explore the call tree of the program and see where time is spent. This makes it much easier to figure out where the performance bottlenecks are in your program.

   You can also use the “Memory” button to collect data about the distribution of object classes used in the program.

3 Deliverables

There are no deliverables for this lab.