CSCU9A5 Practical

Debugging

The purpose of this practical is to familiarise yourself with either the Eclipse or IntelliJ IDEs and use the powerful debugging tools available with these tools. If you are not familiar with either IDE then please use this practical to try them both out and see which one you prefer.

We are going to start by investigating how to use breakpoints in general and then look at different ways to step through your code with a debugger and examine the call stack, values of different variables and means of using break points to work out the best way to pause your code and investigate your problem.

We will use the code shown in the lecture notes as our starting point but you are also welcome to try these principles out on code of your own. Download the *Reviews.java* and *100.txt* files from the Canvas page that you accessed this practical sheet from then create a project in either Eclipse or IntelliJ and add these two files to this project.

The constructor for the Reviews class uses the *loadReviews* method to load a set of product reviews from the file 100.txt into the array list *reviews*. Each line in *100.txt* contains individual reviews that are tab separated and contain a title, a review and a rating at the end of the line. We ignore the title and the rating and just load the text of the review as a String object.

There are two further methods in Reviews.java called printTop and findLongestReview. printTop will print out the first numReviews in the list of reviews and findLongestReview will return the index of the longest review in the list of reviews.

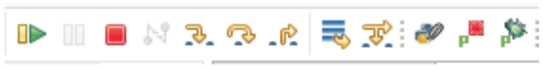
Setting a Breakpoint

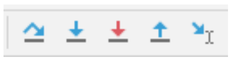
Let's start the debugging process by watching program execution from the very start. Go to line 12 and set a break point by either double clicking on the line number in Eclipse or single clicking to the right of the line number in IntelliJ. You should see a little blue dot appear next to the line number in Eclipse or a red dot in IntelliJ.

Set the program going in Eclipse or IntelliJ by selecting the Run->Debug menu option (you can also just press the little bug icon that is next to the Play/Run icon in either of these IDEs). You should find that the program will pause at the line that you set the breakpoint at and in the case of Eclipse, your IDE windows will be rearranged to help display debugging information more effectively (this is the Debug perspective in Eclipse and can be returned to the default Java perspective by clicking on the little icon in the top right corner that has a J in it). You should leave it as it is for now since we want to see this debugging information. IntelliJ will show the debugging information in a debug tab at the bottom of the IDE.

In either case, look for the variables list which will just contain args since that is all we know about at the point where we paused the code. args are the command line arguments that were passed into the program and they will be empty by default (these are really useful when you want your program to behave differently each time you run it, for example you could pass in the file name containing the reviews we would like to load).

Debug Actions
The debug buttons for Eclipse and IntelliJ are shown below, locate them in your IDE and check what each of them do by placing your mouse cursor over them (but don't click on them):

Eclipse: 

IntelliJ: 

We will now use these buttons to step through our code and will begin by stepping into the execution of line 12 which is responsible for creating an object of type Reviews. This should create a Reviews object and call the relevant constructor.

Step Into
Press the step into button or use the correct keyboard shortcut (F5 for Eclipse, F7 for IntelliJ) and check that you move to the constructor of Reviews, then keep selecting the step into action until you find yourself at line 22.

Call Stack
You should see a call stack on the upper (Eclipse) or lower (IntelliJ) left showing that you have jumped from the main method and are now in the init method (another word for the constructor). For the variables that are visible in the current method, you should see a reference to this (the object you are in) and if you expand it, you should see a reference to its single attribute called reviews. IntelliJ will also show reviews as a separate entity since it is a locally visible value.

Think about what should happen if you select the step into action again and what you expect to see, then step into the code of line 22 and see if the code actually does this.

Step Over
If all goes well, you should find yourself inside the loadReviews method at line 34. This line is going to create an empty ArrayList of String objects called data. Rather than step through this whole process, use the step over function to look at the result of taking this action (keyboard short is F6 in Eclipse and F8 for IntelliJ).

Variable Properties
You should now find yourself at line 40 and should see that the reference data has been initialised to point to an ArrayList object that is currently empty. You should also note that reviewFileName has the contents 100.txt. Try putting your mouse cursor over reviewFileName on line 40 and waiting to see if the IDE shows that it has this value. This is a relatively quick and easy way to check the value of variable if you're not sure what it is.

You should notice that we don't have an entry for the BufferedReader in yet. This is because it has not been assigned a value yet since we have not executed line 40. Try stepping over line 40 and confirm that you can now see the reference for in being added to your variable list. Step over line 41 and check that you can see that line is also now added and that you can see the contents of line both via the variable list and the mouse over action. You will get different levels of detail with each method of inspection with Eclipse tending to me more verbose than IntelliJ.

Now step over line 43, look for the toks variable in your variable list and expand its contents. This should show you the 3 tokens that were extracted when we split our review string using the tab character. The first entry should be a title, the second entry a review and the final entry should be a number. If you now step over line 44, you should see that the content of toks[1] is now referenced in data[0] and that you have gone back to the next iteration of the while loop.

Try stepping through the while loop a few times and checking that successive review entries get added to the ArrayList called data and that the other variables used in this method are assigned the values that you expect them to be.

Step Out
I would guess that you are not going to want to step through loading all 100 reviews in the review file but may want to watch what else happens after we have done this. We can use the step out action to achieve this (F7 in Eclipse and Shift F8 in IntelliJ). This will complete the current method we are in and return us back to the calling method. Try this action and confirm that find yourself back in the original constructor for Reviews and select it again and check that you are now back in the main method.

Resume
If you don't want to trace the action of any further code but want the program to run to completion (or the next available breakpoint), select the Resume action (F8 in Eclipse, variable in IntelliJ depending upon platform) and confirm that your program now finishes by printing the value 73 to the console.

Let's now try jumping between different breakpoints. First place an additional breakpoint at line 64 then start the program in debug mode. This should cause execution to pause at line 12 as before but if you select the resume action, it should now pause again at line 64 in the findLongestReview method. If you select the step over action until you reach line 71, you should observe the local variables be initialised to 0 and added to the variable/expression list. You should also see that we start going round our for loop and will pause with our loop variable r having the value of 0 as well. Try using the step over action for the first 5 iterations of this for loop and watch what happens to maxLength and longest based on the lengths of the lines that we are processing.

Initially you should see that maxLength is updated immediately since our current maxLength is 0. The next line is even longer than the first line so maxLength gets updated again. The third line is however shorter so the update to maxLength and longest is skipped. We will still not have found a longer length of line by the time r = 5 and will probably not want to watch the remaining 95 lines of data get processed so just resume the execution of the code and let it complete its task.

Rather than step through each line of the loop, you would probably be most interested in only pausing when the value of maxLength is changed. Set a breakpoint to achieve this and check that the program only pauses when you have found a longer line. Once you have got this to work, disable or remove the breakpoint so that it does not interfere with the next step. In Eclipse this is achieved by right clicking and selecting the Disable Breakpoint menu option. In IntelliJ, you should be able to right click the breakpoint and uncheck the Enable Breakpoint option.

Conditional Breakpoints
You may be interested in a specific situation that seems to cause problems with your program. Rather than stepping through lots of code, we can use a conditional breakpoint to achieve this.

Create a breakpoint at line 75 and then right click on it and select Breakpoint properties in Eclipse (right clicking the breakpoint in IntelliJ will directly bring up the properties but you will need to then click the More link in the bottom left to see the full set of options).

In either case, ensure the Conditional/Condition box is checked then enter the phrase maxLength==104 into the text box. Now try running your program in debug mode and check that execution only pauses when the variable maxLength has the value 104.

When debugging you use this type of breakpoint to get your code to pause when a variable ends up with a value you think it shouldn't have but can see that it is fact getting set to (a common occurrence indicating a broken assumption). This type of breakpoint allows you to see the exact state of the variables and chain of method calls (via the call stack) that is leading to your program to crash or produce the wrong answer.

You can actually include more complex expressions here if you wish. Try adding a breakpoint to line 71 and instead of using the condition maxLength==104, try putting in the following text and observe what happens:

```
reviews.get(r).contains("bought");
```

This will result in your program pausing only when the current review you are looking at contains a particular word and is very useful if your code is processing data and only seems to break on particular lines containing an identifiable word or phrase.

Watch Expressions
You may want to continually monitor the state of some elements in a data structure while stepping through your code but these values are not being shown by default in your debugger. You can add further expressions to be evaluated in Eclipse and IntelliJ by using the Expressions tab in Eclipse or the text box above the Expression debug variable list in IntelliJ.

For Eclipse, select the Expression tab in the top right and click on the "Add new expression" field. Type reviews.get(r) in the text box and press enter. For IntelliJ, just type this expression in the text box above the watched variables list and then click the + symbol on the right side of the text box.

Now set a normal breakpoint at line 71 and then start the program in debug mode. Use the resume button to keep running until you get to line 71 and observe that the expression you have entered is being correctly updated each time you loop around the for loop.

Debugging Challenge
For the last part of this practical, we are going to try and identify an incorrect assumption that was made when the program was written by using a different data file. Change the data source file name on line 22 to 20.txt and rerun the program. You should find that it will crash. See if you can use the debugger to set a breakpoint in the right place and work out why.

You are welcome to do this in a small group if you wish and can discuss how you can use breakpoints and watch expressions to determine what is happening.