Lab 4: Part 1
Debugging Loops

Last week we explored some basic features of BlueJ’s debugger. We saw how to interpret “ugly red stuff,” to set breakpoints, and to examine variable values. This week we will do a bit more exploration to help you learn how to fix problems that often occur when writing programs that use loops.

For this exercise, we will use a version of a program demonstrated in class that “improves” a sentence by replacing as many of its words with synonyms from the dict.org Moby thesaurus. The version shown in class always used the first synonym provided by dict.org. The version you will be working with today is designed with verbosity in mind. Rather than choose the first synonym, it always chooses the longest. A sample of the output this program might produce is shown below.

To begin, download the IncompleteWordReplacer program from the “Labs” section of the CS 134 web site, unpack the project and open it using BlueJ.

Infinite Loops

Now that you have learned how to write programs that contain loops, you have the ability to write programs containing a very special kind of error, an infinite loop. An infinite loop is a loop that tries to execute forever. When you write a loop, the condition placed in the loop header is supposed to eventually tell the computer to stop repeating the statements in the loop body. Unfortunately, if you make an error writing the condition (or the loop body), you can easily end up with a loop that never stops. In this case, your program will just appear to go into an unresponsive state since it is too busy executing the loop to do anything else. (Many of you were lucky enough to experience this during last week’s lab.)

First, let’s learn another way to stop a program that gets itself into this state without having to use “Reset Java Virtual Machine.”

- Run the LongestSynonyms program, enter a short sentence in the top text area and then click the “Find Synonyms” button. Because the code includes an error that leads to an infinite loop, the program will freeze up (the button will remain highlighted in blue).
- Bring the debugger window forward on your screen. You can do this by first clicking on the BlueJ project window for IncompleteWordReplacer and then selecting “Show Debugger” from the “View” menu. You will notice that the debugger window only displays its control buttons and the thread list. The method invocation lists and variable lists are missing.
- Click once on the “Terminate” button.

This should cause the program to terminate. When a program seems to go into an infinite loop, it is usually reasonable to simply terminate its execution in this way and then take a quick look through its code to see if you can identify the cause of the looping problem directly. With this in mind, let’s take a
quick tour of the program’s `buttonClicked` method. Our goal won’t actually be to identify the cause of the problem, but to become familiar enough with the structure of the program to use the debugger effectively to diagnose the problem later.

If you scan through the text of the `buttonClicked` method you will discover that it contains three `while` loops. Two of them are nested as subparts of the other loop.

The first loop’s header can be found around line 51 of the program’s text. This loop begins with the header:

```java
while ( sentence.length() > 1 ) {
```

(Since any change made to the program will change all the line numbers, the line numbers provided in this handout may be approximate. Any time we refer to a line number, we will also include the text we expect you to find on that line so that you can be sure you are looking at the right code.)

This loop is executed once for each word in the sentence the program is trying to “improve”. Each time this loop executes, the program asks the `dict.org` server for synonyms for the current word. Since the server may send back many lines of synonyms, the program depends on a second loop (at line 77) that starts with

```java
while ( ! serverResponse.equals( "." ) ) {
```

This second loop’s body should be executed once for each line of synonyms the server sends in response to a single word from the sentence. That is, because it is nested within the first, larger loop, this loop will usually execute many times each time the body of the main loop is executed once.

The goal of this inner loop is to join all of the lines received into one big `String` of synonyms separated by commas called `allSyns`.

The third loop in the `buttonClicked` method can be found around line 89. It begins with the line:

```java
while ( firstSynonymEnd >= 0 ) {
```

Like the second loop, it is nested as a subpart of the first loop. After the second loop has placed all of the synonyms in the variable `allSyns`, this third loop should execute once for every synonym found in `allSyns`. While doing so, it keeps track of the longest synonym it has found so far in the variable `longestSyn`. As a result, when it reaches the last synonym, the variable `longestSyn` should be associated with the longest synonym found for the current word. The main loop then adds this synonym to the replacement it is building for the original sentence and moves on to the next word in the original sentence.

Any of these three loops might contain the bug that is causing our program to loop forever. In the last lab, many of you saw that it is possible to make a mistake in a loop that processes lines sent by a server that will cause that loop to never terminate. It is equally possible to make such mistakes in loops that process the individual subparts of a large string. So, to start with we must consider all three loops as suspects. Luckily, we can use the debugger to quickly help us narrow down the possibilities.

To see how to do this, first arrange the windows on your screen so that you can see the project window and the debugger window (if you closed the debugger window, select “Show Debugger” from the “View” menu that appears when the project window is the active window on your screen). Make sure the
debugger window is positioned so that it won’t be covered up by your program window when you run the program. Now:

- Run the `LongestSynonyms` program again, enter a sentence and press “Find Synonyms”. The program should again freeze up.
- Bring the debugger window forward on your screen.
- In the threads list, find your program’s thread, AWT-EventQueue-0. (If you don’t see this thread, use the “Options” menu to un-hide system threads.) Click once on the thread’s name to select it.
- Press the Stop/Halt button at the bottom left of the debugger window. Your program will be suspended and a method invocation list and the variable lists will be displayed.

If you examine the method invocation list, you will probably find that the method named at the top of the list is `buttonClicked`. That should be a nice, familiar method name and it should not surprise you that your program’s loop is somewhere in `buttonClicked`.

Last week we saw that errors are often detected in methods that are part of the libraries associated with Java, rather than in the code of your own program. Similarly, if you use the debugger to stop a program, there is a good chance the program will be interrupted while executing an instruction in a library method rather than in one of the statements of your program. If this is what happens, then if you look a few lines down the method invocation list, you should find a familiar method name, probably `buttonClicked`.

Now, make sure that the window displaying the source code for `LongestSynonyms` is visible on your screen. Then…

- Start to slowly, but repeatedly click the “Step” button.

You will see that BlueJ steps through the instructions of your program line-by-line as you click “Step”. As it does this, it indicates which line will be executed the next time you click “Step” by highlighting the line and placing an arrow on the line.

- Keep clicking “Step” until you reach the header of a while loop

At this point, your computer should be about to execute the header of the while loops that starts with a line of the form:

```java
while (firstSynonymEnd >= 0) {
```

which is the header of this program’s third loop. Because you have used the debugger to “catch” your program executing this loop, you may be tempted to conclude that this loop is the source of the problem. Don’t be so quick to judge! Remember that this loop is nested within a larger loop. It is nested within the loop with the header

```java
while (sentence.length() > 1) {
```

It might be the case that the inner loop will execute many times but eventually stop, allowing the program to continue on to the next execution of the outer loop. If the outer loop is structured incorrectly, it might fail to ever terminate even though the loop nested inside is completely correct.

You could just try clicking the Step button many times to see if the program ever leaves the inner loop. Go ahead. Try it!
Click on the “Step” button a dozen times or more.

No matter how often you click, the computer appears to just repeat the inner loop. Sadly, this doesn’t prove much. You have seen in class how long the lists of synonyms the dict.org server tends to provide. This loop might execute hundreds of times before it stops. Your clicking finger would wear out first.

There is a much better approach.

• Click on the “Continue” button.

This allows your program to resume normal execution. Unlike the slow rate at which it executes instructions when you are clicking on the “Step” button, the machine is now executing thousands or even millions of instructions per second.

• Place a breakpoint on line 54 of the program. You can do this while the program continues to run by clicking the mouse on the line number that appears on the left edge of the window that displays the program’s code. Line 54 should be the first line of the outer while loop. It should look like:

```java
int firstSpacePos = sentence.indexOf(" ");
```

At the rate the computer is executing instructions now, if the inner loop was correct, it should complete and your program should get back to the line with the breakpoint within seconds. Sadly, nothing happens.

• Now, put a breakpoint on line 90. This is the first line of the inner while loop. It should look like

```java
String curSyn = allSyns.substring(0, firstSynonymEnd).trim();
```

This time, the program should halt almost immediately. This is proof that the loop starting at line 89 is the problem.

The key idea here is that while a program is running you can place breakpoints as probes to find out where executing is actually taking place.

You can find out a lot about what is going on (and ultimately what is going wrong) by looking at the contents of the lower right corner of the debugger window. This area of the window displays the contents of the local variables of the current method (buttonClicked). The names we have been discussing — allSyms, firstSynonymEnd, etc. — are all included in these local variables.

As mentioned above, this loop that appears to be causing problems is supposed to be executed once for each synonym in the String variable allSyms. You can see the current value of this variable in the debugger window. It is a very long string of synonyms for one of the words in the sentence you typed in. Each synonym in this string is followed by a comma.

The loop in which your program is stuck is designed to use the indexOf method to find the comma after each synonym and then use its position to extract the synonym from the other using the substring method. The variable firstSynonymEnd is supposed to be set equal to the position of the comma. If indexOf cannot find any more commas, it is supposed to return -1. Therefore, the condition “firstSynonymEnd >= 0” is supposed to make the loop stop once all of the commas have been used to extract a synonym from allSyms.

You may also want to look for the current value of currentWord. This variable is used by the outer loop as it steps through the sentence you typed in when the program is run. Where does the current value of
currentWord appear within the sentence you typed in? Has the main loop made much progress through the words of your sentence?

At the moment, one interesting piece of information is missing from the debugger window. Each time around the loop your program sets the variable curSyn equal to one of the synonyms in allSyns until it reaches the end of allSyns. If you put the breakpoint where we wanted you to put it, you should be just about to execute the line of code that assigns a value to curSyn. Once this line is executed, its value will be displayed. So,

- Press the “Step” button in the debugger window once.

Look at the value of curSyn. Where does this word appear in allSyns (the list of all synonyms)?

- Press the “Step” button a dozen times again. Each time the line that sets curSyn is executed, look at the value for curSyn shown in the debugger window.

By now it should be clear that this loop isn’t making progress. It is stuck on the first word included in allSyns when the loop is supposed to work its way through all the words in this list.

The problem is that the line

```java
    firstSynonymEnd = allSyns.indexOf( "", " ");
```

at the end of the loop is supposed to find the NEXT comma in allSyns. Unfortunately, indexOf always finds the first occurrence of the String it is told to look for. So, it is always finding the first comma.

In class, we have avoided such problems in examples by changing the String we searched before the use of indexOf designed to find the next delimiter. In this case, one could do this by adding an assignment of the form

```java
    allSyns = allSyns.substring( ... );
```

that set allSyns equal to a String obtained by removing the word that was just processed and the comma that follows it from allSyns. Is is up to you to figure out what to type were we placed ... 

Another approach is to use the fact that indexOf can be given a second parameter that determines where it begins its search. In this approach, rather than adding an extra line of code, you would just change the last line of the loop to look like

```java
    firstSynonymEnd = allSyns.indexOf( ",", ... );
```

where the “...” needs to be replaced by an expression that will produce the position of the first letter after the comma whose position was previously used as the value of firstSynonymEnd.

- Pick one of these approaches and make the appropriate change. Test your change by first pressing the “Terminate” button and the compiling and running the new version.

- If necessary, use the debugger to get your change to work!