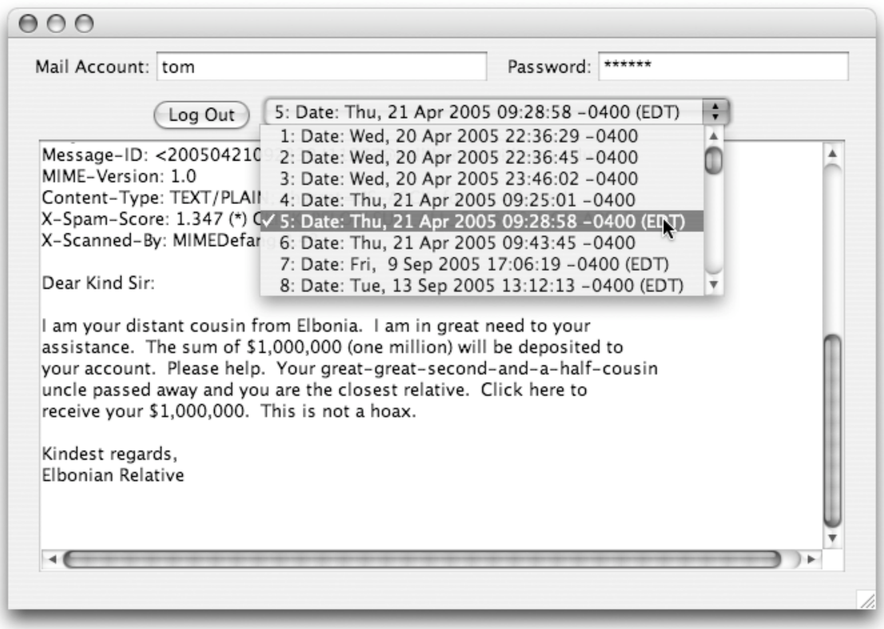
Lab 4: Part 1

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 us learn how to fix problems that often occur when writing programs that use loops.

For this exercise, we will use a slightly different version of our POP client program. A sample of the interface that should be provided by the program you will be debugging is shown below.



Mail Account: tom
Password: ******
Log Out
5: Date: Thu, 21 Apr 2005 09:28:58 -0400 (EDT)

```
1: Date: Wed, 20 Apr 2005 22:36:29 -0400
2: Date: Wed, 20 Apr 2005 22:36:45 -0400
3: Date: Wed, 20 Apr 2005 23:46:02 -0400
4: Date: Thu, 21 Apr 2005 09:25:01 -0400
✓ 5: Date: Thu, 21 Apr 2005 09:28:58 -0400 (EDT)
6: Date: Thu, 21 Apr 2005 09:43:45 -0400
7: Date: Fri,  9 Sep 2005 17:06:19 -0400 (EDT)
8: Date: Tue, 13 Sep 2005 13:12:13 -0400 (EDT)
```

```
Message-ID: <200504210
MIME-Version: 1.0
Content-Type: TEXT/PLAI
X-Spam-Score: 1.347 (*) C
X-Scanned-By: MIMEDefa

Dear Kind Sir:

I am your distant cousin from Elbonia.  I am in great need to your
assistance.  The sum of $1,000,000 (one million) will be deposited to
your account.  Please help.  Your great-great-second-and-a-half-cousin
uncle passed away and you are the closest relative.  Click here to
receive your $1,000,000.  This is not a hoax.

Kindest regards,
Elbonian Relative
```

The innovation in this week's program is that as soon as a user logs in, the program examines all of the email messages available in the user's account and displays a `JComboBox` menu including a short description of each available message (in fact, the short descriptions used will just be the "Date" field for each message). As soon as the user selects an item from this menu, the program will display the full text of the corresponding message in the text area that occupies most of the program's window.

To begin, download the `MessageMenu` program from the "Lab Schedule and Handouts" page 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*. 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.

First, let's learn how to stop a program that gets itself into this state without having to reboot the machine or use "Force Quit."

- Run the `MessageMenu` program and try to log in to the mail server. Because the log in code includes an infinite loop, the program will freeze up (the Log In 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 MessageMenu 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 one 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. If this is not possible, however, you can use the debugger to determine more about the nature of the problem.

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 `MessageMenu` program again and try to log in to the mail server. 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 not familiar. 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 by pressing the stop button, 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 happened, then if you look a few lines down the method invocation list, you should find a familiar method name, probably `buttonClicked`.

To actually see what is going on in our program's code, we have to get the computer to execute enough instructions to complete the invocation of whatever library method we happened to catch it in and to get back to the instructions in our program. To do this:

- Press the "Step Into" button once.
- Press the "Step" button carefully, but repeatedly until `buttonClicked` (or any other method defined within the `MessageMenu` class) appears as the top item in the debugger's method invocation list.

Now, make sure that the window displaying the source code for `MessageMenu` is visible on your screen. Then, continue to slowly, but repeatedly click the "Step" button until you reach the header of a while loop. You will see that BlueJ steps through the instructions of of your program line-by-line as you click "Step". As it does this, it indicates which line it is currently executing by highlighting the line and placing an arrow on the line.

At this point, your computer should be about to execute the header of one of two while loops. Most likely, you are about to execute a line of the form:

```
while ( retrResponse.startsWith( "+OK" ) ) {
```

Otherwise, the debugger should show that you are about to execute a loop that begins with the header

```
while ( ! messageLine.equals( "." ) ) {
```

The second loop appears within the body of the first loop. Its header appears just a few lines after the first loop's header. Our real goal is to start by examining the second loop. So, if you are not already about to execute its header:

- Press the "Step" button repeatedly until the header of the second while loop is highlighted.

The condition in the loop's header test to see if the latest line received from the server consists of a single period, the convention used by the POP protocol to indicate the end of an email message. Let's walk through this loop's slowly once to see what is going on:

- Find the values of the instance variables `messageLine` and `messageBody` within the BlueJ debugger window.
- Press "Step" several more times (more than 5, less than 10), and watch both the lines that are being executed and the changes that occur in the values of `messageLine` and `messageBody`.

It should become clear that this loop is collecting the lines that make up an email message one-by-one as they are sent by the server. Unless the email message is quite small, it will take a long time for this loop to complete if you just keep pressing the "Step" button. So, instead:

- Place a stop sign on the first line after the loop:

  ```
  String messageDesc = "";
  ```

- Press "Continue" once. The computer will execute the statements in the loop until its execution is complete and then stop again when it hits the stop sign you placed after the loop.
- The next few lines are designed to extract the "Date" header from the message, Press "Step" repeatedly but slowly to execute these lines. Observe how the values of the variables used change.
- Stop when you reach the line that sends a "RETR" request to the server.

The comment on this line says that it is to get the **next** message. The actual message retrieved, of course, depends on the number placed after the code "RETR" in the packet sent. This number is determined by the value of the variable `messageNum`. Check the value associated with this variable in the debugger window.

If the loop really is going to fetch the next message when it sends the next "RETR" command, then the value of `messageNum` will have to be increased. We can now quickly verify that this is not happening.

- Click in the channel to place a stop sign on the line that is about to be executed (the invocation of `out.println`).

- Now, click the "Continue" button twice.

It probably looks like nothing happened. In reality, we just executed the entire loop one more time. Clicking "Continue" the first time caused the computer to execute instructions until it hit the stop sign you placed after the loop that retrieved the lines of a message. Pressing "Continue" the second time caused the computer to execute the lines that extract the "Date" header and stop after sending another "RETR" command to the server. So, we can now execute all the instructions in the loop body with just two clicks of the "Continue" button. Do this a few times, and it should be clear that the problem is that we left out the statement needed to increase `messageNum`.

Press "Terminate" to end the execution of the infinite loop. Then fix this problem by adding the line

```
    messageNum = messageNum + 1;
```

between the statement that invokes `out.println` and the comment that precedes it.

Run the program again and you should see that the loop now completes, filling the header menu properly.

In general, when debugging what appears to be an infinite loop, you should begin by using the debugger to stop the execution of your program so that you can determine which loop within the program is executing. If examining the body of this loop does not quickly reveal the source of the problem, observe the values associated with the variables used within the loop as you repeatedly press the "Step" button. If the loop is large or contains a nested loop, it may help to place stop signs within the loop's body and repeatedly press "Continue" instead of "Stop".