Lab 3

A Multi-Message Email Reader

Due: Wed. 9/25 at 11PM (for Mon. aft.), Thurs. 9/26 at 5PM (for Mon. evening), or Thurs. 9/26 at 11PM (for Tues. aft.)

The goal in this week’s lab is to exercise your new knowledge of if statements by modifying your mail reading program so that it provides a more flexible and user-friendly interface. Then, at the end of the lab we will have you experiment a bit with another new type of statement — the while loop.

The type of window that should be displayed by this new version of the program is shown below.

There are now 3 buttons: Login, Logout, and Get Message.

• When the user clicks the Login button, your program should connect and login to the mail server.
• When the user clicks the Logout button, your program should send a QUIT message to the POP server and close the connection.
• In between, the user may click the Get Message button as often as desired. When the user clicks the Get Message button, your program should retrieve the requested message and display the message’s most significant header lines and its entire body in the upper text area. The text area should be cleared before the new message is displayed so that only one message appears in the window at a time. Notice that with this interface it is not necessary to login for each individual message retrieved.
• When the program starts and after Logout is pressed, only the Login button should be enabled. When the user is correctly logged in, the Login button should be disabled and the Logout and Get Message buttons should be enabled.
• Finally, if the user makes a mistake, your mail program should display a message expressed in English that is free of any protocol specific codes (such as “-ERR”) in the message text area (i.e., the upper text area).
• All +OK and -ERR responses sent by the server should be displayed in the lower text area.
Getting started
You should start with a copy of the code you wrote last week. To do this, use the Finder to locate the folder containing your Lab 2 project. Double click on the package.bluej file to open the project in BlueJ. Click the update button in BlueJ after you open the project to make sure you are working with the latest copy of your code. If you are using a different computer, you will need to checkout Lab 2, as you did last week. (For instructions, see http://www.cs.williams.edu/~cs134/submission2.html.)

Once Lab 2 is open, use the Team->Checkout option in BlueJ to checkout Lab 3 from our server. At this point, you should have two BlueJ project windows open. Arrange them on your screen so you can see both of them at once. Check to make sure one says “lab2” at the top, and the other says “lab3.” Please be very careful with this next step! Double click your code from last week in your Lab 2 window to open the BlueJ text editor. Use Command+A to select all of the code, and then use Command+C to copy it. Now go to your Lab 3 window. Click the New Class button to make a new GUIManager class, and name it the same name as your code from Lab 2. Double click on this new file in Lab 3 to open it. Use Command+A to select all of the code, and then click delete to remove it. Use Command+V to paste your code from last week into this new file. Click compile in Lab 3 to make sure your code still works.

Close your lab2 window so you don’t modify last week’s work. The remainder of your lab should be done in the lab3 BlueJ project. Before going any further, make sure you can commit/push the code in your lab3 project. Always commit/push your code at the end of your lab session, even if you aren’t finished with the lab.

IF YOU DO NOT HAVE A COPY OF YOUR LAB 2 CODE IN A NEW BLUEJ PROJECT NAMED LAB 3, OR IF YOU CANNOT COMMIT/PUSH YOUR CODE, GET HELP FROM A TA OR AN INSTRUCTOR BEFORE GOING ANY FURTHER.

Implementation plan
1. Build the User Interface: Add code to your constructor to display the additional JButtons that are part of the new interface. You should associate an instance variable name with each of the three buttons because you will need to know which button is clicked in the buttonClicked method.

2. Handle the 3 Buttons Differently: Modify the buttonClicked method to behave differently depending on which button was clicked. To do this, you need to include the declaration of a JButton parameter in the method’s header:

   public void buttonClicked( JButton whichButton )

Within the method, make a three-way choice using a pair of nested if-else statements to decide which button is clicked and execute only those statements relevant for that button. The code that creates the NetConnection and sends the USER and PASS commands should be executed when the Login button is pressed, the code to send the RETR command and display the server’s response should be executed when Get Message is pressed, and code to send a QUIT command and close the connection should be executed when Logout is pressed.

In last week’s lab, you created a new NetConnection each time buttonClicked was executed. This week, you will only create a new connection when the user clicks the Login button. Other executions of buttonClicked will use the NetConnection created the last time Login was clicked. As a result, while you used a local variable declared within buttonClicked to refer to the connection last week, this week you will need to change that variable’s declaration to make it an instance variable.

Note: If you took the time in last week’s lab to implement the optional check of whether the server sends +OK or -ERR in response to the PASS command, separating your code into the three sections described above is a bit trickier. The next paragraph discusses how to proceed. Skip the next paragraph and go to step 3 if you only used an if statement to check the result of the RETR command last week.
If you added the optional if statement last week, it is a bit harder to separate the code from last week into the three sub-sections described above because the second if statement you wrote last week is clearly associated with the USER and PASS commands but contains the code related to the sending of the RETR command. We suggest you proceed as follows. First, put all of the code from last week except the code to send the QUIT command and to close the connection in the branch of the three-way if that handles the Login button. Next cut the code that sends the RETR command and handles the server’s response out of the true-branch of the second if statement you added last week and instead place it in the branch of your new three-way if that handles the Get Message button. This may leave you with an if statement one of whose branches is empty, but don’t worry. You will fill it again in step 4. Finally, place the code to send the QUIT command and close the connection in the branch of the three-way if that is associated with the Logout button.

3. **Enable/disable the Buttons:** Add code to enable and disable buttons appropriately. Initially, only the Login button should be enabled. Once the user logs in, the Login button should be disabled and the other two buttons should be enabled. Then, when the Logout button is pressed, the buttons should return to their initial state.

4. **Detect Login Failures:** When the login button is clicked, your program should only change which buttons are enabled if the login attempt is actually successful. If during last week’s lab you wrote the optional if statement to check the server’s response to the PASS command, this will be easy. You simply need to put the code to change the button settings in the branch of the if statement that corresponds to a +OK response from the server. Otherwise, you will need to write an if statement now to check the server’s response to your PASS command. You only need to check the response to the PASS command. The server will always respond “+OK” to a USER command, even if the account provided does not exist. If the response to PASS starts with “+OK”, you should disable the Login button and enable the other buttons. Otherwise, you should display an appropriate error message in the upper text area and leave the buttons as they are.

If a login fails, your program should immediately send a QUIT to the server and close the connection.

5. **Replace nextPOPResponse with a Loop:** Just as we did with the if statement last week, we want to introduce you to a new construct in this lab to better prepare you for our discussion of its use in class. The construct is called a while loop. The while loop provides a way to tell Java to execute a collection of statements repeatedly. In last week’s lab, you used a method named nextPOPResponse to fetch the lines of an email message that the server sent in response to an RETR command. We would like you to replace the command that used nextPOPResponse with while loops that use nextLine repeatedly to access all of the lines in an email message.

The section entitled “Input Loops” below provides the details you will need to write the type of while loops required in this program. Using the information provided in that section, add while loops to perform the work previously done using nextPOPResponse.

6. If you want a little more programming practice, consider adding some of the “Bells and Whistles” described later in this handout. Then, submit your work by following the instructions at the end of the handout.

**Input Loops**

There are many situations in which a program requires executing a series of steps repeatedly until some desired state is reached. Java includes several statements designed for such situations. One is called the while loop. Syntactically, a while loop is similar to an if statement with no else part.
That is, just as you can write:

```java
if ( condition ) {
    a list of statements that may or may not be executed
}
```

you can write

```java
while ( condition ) {
    a list of statements that may be executed repeatedly
}
```

When the execution of a program reaches a while loop, the computer first checks to see if the condition specified currently holds. If not, it skips the list of statements provided in the body of the loop just as an if statement would skip the statements it controlled. If the condition in the while loop is true, however, the computer executes the list of statements provided in the body of the loop and then it repeats the whole process. That is, it again checks to see if the condition is still true, and then either skips the statements in its body or executes them a second time and then repeats the process again.

We would like you to replace the line in your program that uses nextPOPResponse with a pair of while loops that perform the same function as the nextPOPResponse line with a little twist. The twist is that your final program should not display all of the header lines the server sends. Instead, it will just display the most interesting headers — “From:”, “Date:”, and “Subject:”. To familiarize you with the use of a while loop, we will first explain in detail how to construct a single loop that will display the entire email message just as you did last week using nextPOPResponse. Then, we will guide you to replace this single loop with a pair of loops that display the message headers more selectively.

### A Simple Input Loop

Recall that the POP server tells your program that it has finished sending all the lines of a requested email message by sending a line consisting of just a single period. You can therefore retrieve and display all of the lines in a message using just `in.nextLine` by writing a loop that retrieves and displays the lines of a message one after another until it reaches a line containing just a period.

Like the if statement you wrote last week, such a loop will need to use a variable name to refer to the most recent line of the message retrieved from the server. This name can then be used to append each line received to one of your JTextAreas and to check whether the latest line sent from the server contains just a period, indicating that it is the last line. In particular, if `lineReceivedFromServer` is the name of this variable, the condition:

```java
! lineReceivedFromServer.equals( "." )
```

can be used as the condition to tell Java when your loop should stop executing. The ! at the beginning of the condition is interpreted as “not”. Such a condition states that the loop should be executed as long as the last line received does not equal a single period.

To use this condition, you will have to execute an assignment statement that associates `lineReceivedFromServer` (or some other name of your choice) with the line most recently retrieved from the server. This assignment statement, together with an invocation to append the contents of the line to your program’s large text area, will perform the two operation you need to perform for each line the server sends. Therefore, you will want your loop to repeat these two commands over and over again.

The tricky part is that the computer will check the condition of your loop before executing its body for the first time. If you ask the computer to check a condition involving a variable for which no assignment statement has been executed, your program will be terminated with an error. To avoid this, you have to retrieve the first line of the email message using an assignment statement that is placed just before the
loop. The action performed by this assignment, however, is also one of the things you want done repeatedly. Therefore, even though such an assignment belongs before the loop, it will be necessary to also have a copy of the assignment in the loop to make sure it is repeated. As a result, the code within your loop must first process (i.e. display) the previous line received from the server and then retrieve the next line. Thus, the lines to process an email message sent by the server will be:

```java
lineReceivedFromServer = toServer.in.nextLine();
while (! lineReceivedFromServer.equals(".") ) {
    message.append( lineReceivedFromServer + "\n");
    lineReceivedFromServer = toServer.in.nextLine();
}
```

The first time the loop’s body is executed, it will display the line retrieved by the assignment statement that precedes the loop. Every other time the loop is executed, it will display a line that was retrieved by the execution of the assignment as the last step of the previous execution of the loop body.

Please replace the line in your program that uses nextPOPResponse with a loop of the form above. Before running your program, consider the following questions. Will this code display the first line received from the server? Will this code display the last line received from the server (the period on a line by itself)? Now, compile and run the program to see if it works as you expected. The final result should look like the image shown below. (Note that this version shows the period at the end of the message. You may exclude that if you prefer.)

Separating the Headers from the Body
Recall that the text of any email message a POP server sends in response to a RETR command can be divided into (at least) two subparts. First, the server sends a sequence of header lines (each of which starts with a word followed by a colon). Then the server sends the body of the message. These two parts of the email message are always separated from one another by a single blank line. As your next step we would like you to revise your program to use one loop to retrieve and display the headers and another to retrieve and display the message body. To make it clear that each loop is processing the desired portion of the message, these loops should be designed to display all of the headers in your program’s lower text area and all of the lines of the message in the upper text area.

Please modify your program to accomplish this. The process should be fairly simple. The loop given above reads lines until it sees a line containing a single period. Place a second copy of this loop before the original and modify it so that it retrieves lines until it sees an empty line. (Hint: An empty line will “.equals” the String described by placing two quotes "" right next to one another.) Finally, modify the
new copy to display lines in the lower text area. This splitting of the parts of the email between your two text areas is intended as a temporary change to give you visual evidence that each loop is processing the appropriate portion of an email. The result is shown to the right.

Filtering the Headers
Next, we want you to modify the loop that displays the headers to only display the most interesting headers. The key here is to realize that just as we have been able to place if statements within if statements, Java also allows a programmer to put if statements within while loops. (In fact, Java also allows while loops within while loops and while loops within if statements and just about any other form of nesting you can imagine.)

Modify the loop in your program that retrieves and displays header lines by adding a nested if statement in such a way that the loop will still retrieve all the header lines but only display lines that begin with the prefix “From:”. (There should only be one such line in each message.)

Once this is working, modify the loop by either inserting more if statements by using the logical “or” operator (||) so that the loop displays all headers that start with “From:”, “Date:” or “Subject:”.

Finally, recall that displaying the headers in the bottom text area was intended as a temporary change to make it clear which of you two loops was processing which lines. Now that it is working, change your first loop so that it displays whatever lines it chooses to display in the program’s upper JTextArea so that they appear just before the message body. The final result is shown below.

Bells and Whistles
The functionality described above is all that is required for this lab, but if you have the urge to add extra features to the program you have created, there are many options. If you have the time, working on such extras is a great way to make sure you have fully mastered the material covered by each of our labs.

With this in mind, we provide some inspirational suggestions below. While we encourage you to implement some of these optional features, we want to make it clear that you should not do so before you have completed a fully functional version of the basic program required for this week’s lab. In addition, we recommend that you make a copy of your working version before you try to incorporate any of the op-
tional features. (In BlueJ, if you Control+click on your program, you can choose duplicate. Just pick a new name to avoid confusion.) If your efforts to add any of the extended features fail, you will then have the working copy of the basic program as a fall back.

Unlike the other sections of this handout which try to provide all the information needed to complete the basic assignment, the suggestions below are intended just as helpful hints. They will get you started, but may not provide all the necessary details. If you pursue these or any other extensions, talk to your instructor for additional guidance.

1. There are many cosmetic changes you can make to give your program’s interface a more professional feel. With most programs, pressing the return key serves as a shortcut for pressing some default button. In this program, for example, one might expect pressing return to be equivalent to clicking the login button until the user has logged in and to be equivalent to pressing “Get Message” while the user is logged in. Just as you can respond when a button is clicked by putting appropriate code in a buttonClicked method, you can respond when the user presses return by putting appropriate code in a method of the form:

   ```java
   public void textEntered( JTextField whichField ) {
      ...
   }
   ```

   The JButton class includes a handy method named doClick that you are likely to want to use in your textEntered method. It tells a button to act as if it was clicked. When this method is invoked on a button, that button will change color and the buttonClicked method will be executed just as if the button had actually been clicked.

2. A more subtle improvement to your interface would be to ensure that appropriate components are automatically “activated” as the user works. For example, in your program it is probably the case that after you enter a message number and click “Get Message” if you want to fetch a different message you have to click in the message number text field before you start typing the new message number. It would be nice if that text field was automatically activated after a message was fetched so that you could fetch the next message by simply typing its number followed by return (assuming you implemented the previous suggestion). JTextFields provide a method named requestFocus that can be used to implement this behavior. When a text field is told to request the focus, it becomes active as if the user had clicked on it.

3. If you really want to redesign your interface completely, there are two methods you may find interesting. Just as one can add components to the contentPane, you can also remove components. This can either be done one component at a time by passing the component to be removed as an argument in an invocation of the form

   ```java
   contentPane.remove( ... );
   ```

   or you can remove everything by invoking contentPane.removeAll(). For example, when a user logs in you could remove the login components instead of just disabling the login button. Another handy trick is to resize the program’s window using

   ```java
   this.setSize( newWidth, newHeight );
   ```

   If you use some of these methods and things are not quite working, ask us about validate.
Submission Instructions

Take a final look! When your program seems to be working correctly, take the time to test it thoroughly. See how it behaves when you do unexpected things like leaving text fields empty or entering invalid message numbers. After you are confident that your program is correct, you should take a few extra minutes to look over the code before turning it in. Look carefully for any errors that might exist but not have been serious enough to cause your program to malfunction during your testing.

Next, look carefully at your programming style. Failure to follow the style suggestions below may result in a lower grade for your programming work.

Make sure your code is formatted in a way that makes it easy to read. Blank lines should be used to separate distinct components of your program from one another. Indentation should be used to distinguish relationships. For example, the instructions that make up the body of a method should all be indented by the same amount and they should be indented more than the header. If you select the “Auto-layout” item from the BlueJ “Edit” menu, BlueJ will automatically format your code in a reasonable way. You may want to fine tune BlueJ’s formatting, but what “Auto-layout” produces is usually a very good starting point.

Make sure the names you chose for your variables help clarify the functions of those names. Avoid short, cryptic names.

Include final instance variable declarations for names associated with values that determine things like the width of program text areas and the port number to which you connect. Use these names in place of their values in the bodies of your constructor and method definitions.

Make sure that you include comments that explain the purposes of the instance variables that you declare. Also provide a comment describing what each method does. If a particular method contains many lines, try to break the body of the method into groups of related instructions and place an explanatory comment before each group. Figures 4.5 through 4.11 in “Programming with Java, Swing, and Squint” provide examples of good formatting and commenting.

Run your program again to make sure it still works after any changes you made while polishing it up.

Now you should be ready to submit your program. You can find instructions describing how to do this on the “Labs” page of the course web site, or at