

Lab 3

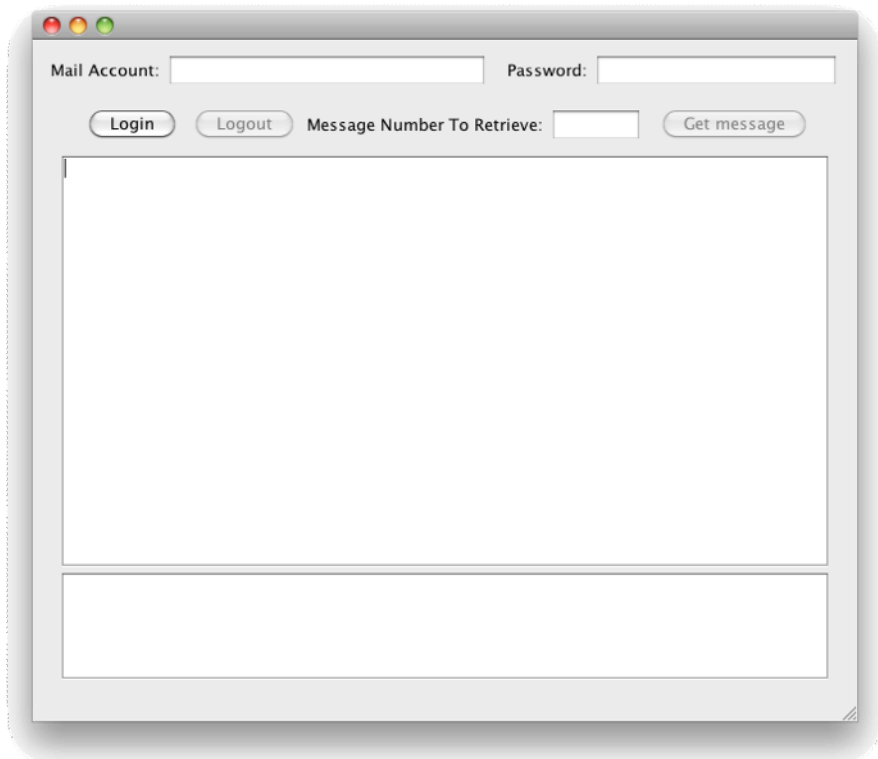
A Multi-Message Email Reader

Due: February 24/25, at 5 or 11 PM

During last week's lab, you constructed a program that could be used to retrieve email messages from a POP server. The interface provided by that program was kept deliberately simple to compensate for the limited set of programming tools we had covered in class before the lab. All of the program's actions were controlled by a single button. In addition, the program gave very little feedback that would be useful to the typical user of an email program.

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

There are now 3 buttons in the program's window: 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 it 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.



While your program will always display three buttons, there are restrictions on the sequence in which the buttons should be pressed. For example, initially the only thing the user should be allowed to do is enter account information and press the Login button. Pressing either the Logout or the Get Message button before logging in would clearly be incorrect. The program should ensure that the user cannot make such mistakes by selectively enabling and disabling buttons so that only appropriate button clicks are possible. When the program first starts, only the Login button should be enabled. Whenever 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 provide feedback that is more informative than displaying a `-ERR` message. Instead, a message expressed in English that is free of any protocol spe-

cific codes should be displayed in the message text area (i.e. the upper text area). When the program is complete, the lower text area will be unused and can be removed from the interface.

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. Select the Lab 2 folder then use the Duplicate command in the File menu to make a copy of it. Now, rename your project with a name identifying it as Lab 3 and including your name. For example, if your name is Floyd you might name the lab FLOYDLab3. Also, remember the name should **not** include any spaces.

When you open BlueJ, it will show your lab from last week. Use the Open Project item in the Project menu to find and open your Lab 3 project. After opening the Lab 3 project, close your Lab 2 project.

Implementation plan

Last week's lab was designed to rely entirely on the `dataAvailable` method to retrieve lines sent to your program by the server. That is, the only invocation of `in.nextLine` within the program should have been within the `dataAvailable` method. During this lab, you will change the structure of your program to more clearly reflect the back-and-forth nature of the dialogue between the server and your client. Instead of relying on the `dataAvailable` method to invoke `in.nextLine`, you will place invocations of `in.nextLine` immediately after the instructions in `buttonClicked` that transmit the requests that cause the server to send lines to your program. As a result, your final version of `buttonClicked` will resemble the code in Figure 4.11 of "Programming with Java, Swing, and Squint" rather than the code in Figure 4.8. When you are all done, the `dataAvailable` method will be removed from your program.

This is a significant change in the structure of your program. As a result, even though you will start the lab with your working program from last week, the implementation plan suggested below will very quickly lead you to produce a program that does not actually work correctly. Don't worry! When you are all done, everything will work as desired.

The plan:

1. 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.
2. Modify the `buttonClicked` method to behave differently depending on which button the user clicks. To do this, you need to include the declaration of a `JButton` parameter in the method's header as in:

```
public void buttonClicked( JButton whichButton )
```

Within the method, use `if` 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 should be executed when Get Message is pressed, and code to send a "QUIT" command should be executed when Logout is pressed.

When you do this, the invocation of `addMessageListener` from last week's code should end up in the code that handles the "Get Message" button. As a result, your program will no longer work quite correctly. You are not allowed to add a message listener to a `NetConnection` if you have added one already. If you press the "Get Message" button several times in a row, your program will try to add itself as a message listener redundantly and you will receive an error message. We will address this issue in a few steps. For now, just check that you can login, get **one** message, and then logout.

3. Add code to enable and disable buttons appropriately. Initially, only the Login button should be enabled. Once it is clicked, it 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. (You do not need to use a boolean variable when you write this code. You may if you want, but it is not required.)

4. Eventually, we want your program to replace the cryptic `-ERR` responses it receives from the server with more informative messages a typical user may understand. This is difficult to do if all of the responses received from the server are processed in the `dataAvailable` method. Within this method, it is difficult to tell whether a particular response was sent as a reaction to an attempt to log in or in response to an attempt to retrieve a message. Accordingly, as your next step, you should reorganize your program so that it does not invoke `addMessageListener` until just after it sends an “RETR” command to the server and receives a `+OK` response to that command. To do this, you will have to add code in your `buttonClicked` method to retrieve and display the `+OK` (and `-ERR`) messages received from the server in response to creating a connection and sending “USER”, “PASS”, and “RETR” commands.

Note: We deliberately left the “QUIT” command out of this list. The server always responds to a “QUIT” with a `+OK`. Therefore, to keep things simple you can immediately close your `NetConnection` after sending a “QUIT” without retrieving the response.

Your program will still have the problem described in step 2. Trying to retrieve two messages in a row will cause your program to add itself as a message listener repeatedly producing an error message. Therefore, make sure not to do this until you complete step 6.

5. Now, using an `if` statement, revise the code in `buttonClicked` that handles the Login button so that it checks the response the server sends after receiving the “PASS” command. If the response 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.
6. To make it possible to retrieve multiple messages, you should change your program so that it never invokes `addMessageListener`. This means you need another way to process the unpredictable number of lines a server may send to you after your program requests an email message using an “RETR” command. To do this you will use a construct called a `while` loop. We have not covered this construct in class yet. Just as we did with the `if` statement last week, we want to introduce you to `while` loops in lab to better prepare you for our discussion of their use in class. The section entitled “Input Loops” below provides the details you will need to write the type of `while` loop required in this program. Using the information provided in that section, you should remove the invocation of `addMessageListener` from `buttonClicked`, remove the `dataAvailable` method from your program, and replace their combined functionality with a `while` loop.
7. The final step is to make your program respond with an appropriate error message if a user tries to access a message but enters a message number that is out of range or invalid. To do this, you should add one more `if` statement after the code that sends “RETR” requests. This code should check the response received from the server and either display an error message or execute the loop that retrieves the message depending on whether or not the server detected an error. Once this is complete, you may want to remove the lower text area to simplify your program’s interface.

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 a statement designed for such situations. It is called the `while` loop. Syntactically, a `while` loop has quite a bit in common with an `if` statement with no `else` part. That is, just as you can write:

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

you can write

```
while ( condition ) {
    a list of 1 or more statements to execute 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 statement provided in the body of the loops just as an `if` statement would skip the statements it controlled. If the condition in the `while` loop is `true`, however, it 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 `true`, and then either skips the statements in its body or executes them a second time and then repeats the process again.

`while` loops can be used to process multiple lines of data received from a server by repeatedly executing the statements required to process a single line. For example, in the WHOIS protocol client shown below, we use the `dataAvailable` method to process the lines of data received:

```
public class WHOISClient extends GUIManager {
    //Size of the program's window
    private final int WINDOW_WIDTH = 600,
                    WINDOW_HEIGHT = 600;

    // The name to request information about
    private JTextField query = new JTextField( 20 );

    // Used to display the information requested
    private JTextArea results = new JTextArea( 20, 40 );

    // The connection to the server
    private NetConnection toServer;

    // Display field, a button, and a text area;
    public WHOISClient() {
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

        contentPane.add( new JLabel( "Name to look up:" ) );
        contentPane.add( query );
        contentPane.add( new JButton( "Request Info" ) );
        contentPane.add( new JScrollPane( results ) );
    }

    // Request and display information from the server
    public void buttonClicked() {
        toServer = new NetConnection( "dns411.com", 43 );
        toServer.out.println( query.getText() );
        results.setText( "" );
        toServer.addMessageListener( this );
    }

    public void dataAvailable() {
        results.append( toServer.in.nextLine() + "\n" );
    }
}
```

We can revise this program to use a loop by replacing the line that invokes `addMessageListener` with the `while` loop shown below:

```
while ( toServer.in.hasNextLine() ) {
    results.append( toServer.in.nextLine() + "\n" );
}
```

The body of this loop is just the statement that had been used to handle lines received from the server in the original program's `dataAvailable` method. The condition in this loop uses a method named `hasNextLine`. The resulting condition holds as long as there are still lines coming from the server. It becomes `false` once the program has retrieved all the lines sent by the server before the server closed its end of the network connection.

We would like you to use a while loop to process the lines of email messages your program receives from the POP server. In your mail program, however, you cannot use the `hasNextLine` method to describe the condition that determines when your loop should stop. The POP server does not close its connection when it finishes sending your program the last line of an email message. It does not close its connection until your program sends a "QUIT" command. Instead, the POP server tells your program that it has finished sending all the lines of a message by sending a line consisting of just a single period. Therefore, the condition you should use will have the form:

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

The `!` at the beginning of the condition is interpreted as "not". This condition states that the loop should be executed as long as the last line received does not equal a single period (assuming that the variable named `lineReceivedFromServer` is associated with the text of the last line from the server).

To use this condition, you will have to execute an assignment statement that associates `lineReceivedFromServer` or some other name of your choice with each line 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 form the body of your loop.

The tricky part is that the computer will check the condition of your loop before executing its body for the first time. Your loop's body will contain an assignment to associate a variable name with a line retrieved from the server, but this assignment will not be executed for the first time until after the loop's condition is checked. 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 since the computer will not be able to interpret the condition in any reasonable way.

To avoid this, you have to retrieve the first line of the email message using an additional assignment statement that is placed just before the loop. Thus, the overall structure of the lines to process an email message sent by the server will be:

```
lineReceivedFromServer = toServer.in.nextLine();
while ( ! lineReceivedFromServer.equals( "." ) ) {
    statements to process a single line
    lineReceivedFromServer = toServer.in.nextLine();
}
```

Enter such a loop in your program to complete step 6 of our implementation plan.

Submission Instructions

Take a final look! When your program seems to be working correctly, take the time to test it thoroughly. Make sure to 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.

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 re-

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 press the “Format” button at the top of the BlueJ editor window holding your program’s code, BlueJ will automatically format your code in a reasonable way. You may want to fine tune BlueJ’s formatting in some ways, but what “Format” produces is usually at least a very good starting point.

Make sure the names you chose for the variables used in your program help clarify the functions of those names. Avoid short, cryptic names. Include final instance variable declarations to associated names with the values used to determine things like the width of program text areas and the port number to which you connect. Use these name in place of the 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. Make sure to include a comment before your class header that includes your name and lab section. Figures 4.5 and 4.11 in “Programming with Java, Swing, and Squint” provide examples of what good formatting and commenting might look like.

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

Now, return to the Finder and look in your “Documents” folder. Find the folder that BlueJ created for your project. Its name should be the one you picked for your project (something like FloydLab3).

- Click on the Desktop, then go to the “Go” menu and select “Connect to Server.”
- Type “cortland” for the Server Address and click “Connect.”
- A window will come up which says “Connect to the file server cortland.”, **select Guest**, then click “Connect.”
- A window will appear where you should select the volume “Courses” to mount and then click “OK.”
- A window will come up which says “You are connected . . .” Click “OK.”
- A Finder window will appear where you should double-click on “cs134”,
- Drag your project’s folder (whose name should look like FloydLab3) into either “Dropoff-Monday” if you are in Monday’s lab or “Dropoff-Tuesday” if you are in Tuesday’s lab. When you do this, the Mac will warn you that you will not be able to look at this folder. That is fine. Just click “OK”.
- Log off of the computer before you leave.

After you have turned in the lab, please complete the web-based quiz for the week. You should use the same web link and password you received by email from our Swedish colleagues last week (the user name is your Williams email address). Please complete the quiz before Friday morning.

The purpose of these quizzes is to clarify both for you and for your instructors whether you have understood the concepts covered during the week. Your quiz score will not count in any way when we compute your final grade for the course. If you do poorly on the quiz, you can take it again as many times you like. If you do poorly even after multiple tries, you should seek extra assistance to understand these concepts.

If you are in an afternoon lab, you can submit your work up to 11 p.m. two days after your lab (11 p.m. Wednesday for those in the Monday Lab, and 11 p.m. Thursday for those in the Tuesday Lab). If you are in the Monday evening lab, your work must be submitted by 5p.m. the following Thursday. If you submit and later discover that your submission was flawed, you can submit again. The Mac will not let you submit again unless you change the name of your folder slightly. It does this to prevent another student from accidentally overwriting one of your submissions. Just add something to the folder name (like the word “revised”) and the re-submission should work fine. We will grade the latest submission made before the

deadline, unless we receive an email indicating that you wish to use some of your “late days” for this assignment.

Grading

Programming labs will be graded on the following scale:

- A+** An absolutely fantastic submission of the sort that will only come along a few times during the semester.
- A** A submission that exceeds our standard expectation for the assignment. The program must reflect additional work beyond the requirements or get the job done in a particularly elegant way.
- A-** A submission that satisfies all the requirements for the assignment --- a job well done.
- B+** A submission that meets the requirements for the assignment, possibly with a few small problems.
- B** A submission that has problems serious enough to fall short of the requirements for the assignment.
- C** A submission that is significantly incomplete, but nonetheless shows some effort and understanding.
- D** A submission that shows little effort and does not represent passing work.

Completeness / Correctness

- GUI layout
- Connecting to the server
- Retrieving mail messages
- Disconnecting from the server
- Displaying mail messages
- Enabling and disabling buttons
- Meaningful error messages

Style

- Commenting
- Good variable names
- Good, consistent indentation
- Good use of blank lines
- Removing unused methods