Lab 2
A Minimal Email Reader

Due: Wed. 2/17 at 11PM (for Mon. aft. lab), Thurs. 2/18 at 5PM (for Mon. evening), or Thurs. 2/18 at 11 (for Tues. aft.)

**Goal:** This week’s lab assignment is to write an email client program that provides one half of the functionality of a typical email client. It will let its user read mail but not send mail.

**You should read the entire handout carefully before coming to lab.** In addition, *before your lab period* you should sketch out on paper the Java code you will need for the first steps of the lab.

**A POP Client Interface**
Your program should display an interface like that shown below.

![Email Client Interface](image)

Across the top of the window should be two textfields in which the user enters his or her account information. Below these should be a field to hold the number of the message the user wants to see. This field is followed by a button labeled “Get Message.” The remainder of the window is filled with two `JTextArea`es. The upper text area is used to display the latest message retrieved. The lower text area will be used to display the “+OK” messages the POP server sends in response to the commands it receives from your program.

Each time the user presses the "Get Message" button, your program will log on to the CS department's email server using the "USER" and "PASS" commands, retrieve the specified message using the "RETR" command, and then log out using the "QUIT" command. In order for this to work, the user will need to have entered all relevant information in the three text fields.

Note that your POP client will work differently from the Thunderbird program you explored last week. Thunderbird fetches all new messages at once. As indicated above, you should retrieve only one message for each button press.
Implementation plan
It is a good idea to implement a program in small steps, testing the correctness of each step as completely as possible before moving on to the next. Otherwise, it can be very difficult to pinpoint errors when they occur. With this in mind, here is a plan you may wish to follow:

1. **Create the User Interface:** Write the declarations and instructions to construct the elements of the desired user interface. This code should resemble several examples you have seen in class or in the text. To ensure that your password is not visible while you are running your program you should use a JPasswordField rather than a JTextField in your program’s interface. (A brief discussion of how to use a JPasswordField is provided later in this handout.)

   Run the program (as described later) to verify that the interface appears as desired. In fact, we suggest that you run the program as soon as you have written enough instructions to create the first few elements of the interface. **Do so repeatedly as you add more components!**

2. **Create the Network Connection:** Add statements to create a NetConnection. Recall that a POP server sends a welcome message as soon as a client establishes a connection with it. So, in addition to creating the connection you should write code to retrieve and display this message in your program’s lower text area. To complete this step you will need to:
   a. Declare a local variable to refer to your NetConnection.
   b. In the buttonClicked method, be sure to include an assignment statement that constructs the connection and associates it with the variable name.
   c. After the assignment that creates the NetConnection, place an instruction using nextLine to retrieve a line from the NetConnection and display it in the program’s smaller text area. Don’t forget to add a new line character (i.e., a “\n”) to the end of the line. Your final code is likely to resemble the code in Figure 4.8 of “Programming with Java, Swing, and Squint.”

   Once you have completed these steps, run your program. If everything is correct, a message that begins with “+OK” should appear in your program’s window.

3. **Terminate the Network Connection:** Next, add a command to send a “QUIT” command to the server. This should cause the server to terminate the network connection after first sending back another “+OK” line. Therefore, the command that sends the “QUIT” command should be followed by a command to retrieve another line from the NetConnection and display it in the program’s smaller text area. Programs that use NetConnections should close the connections once they are no longer needed. Therefore, you should invoke the close method on your NetConnection right after retrieving the server’s response to your “QUIT” command. Run your program and make sure that an extra “+OK” line appears.

4. **Send Commands to Log in:** Now that the connection is being constructed and closed correctly, you can write instructions to login by sending the user’s account identifier and password in “USER” and “PASS” commands. Again, after sending each of these commands you should retrieve the server’s response and display it in your program’s lower text area. These commands should all fall between the commands to make the connection and the commands to send “QUIT”.

5. **Retrieve a Message:** The interesting part, of course, is actually retrieving the email message. To do this, you should add code to send a “RETR” command containing the message number entered by the user to the server. The command to do this should look very much like the lines that send the “USER” and “PASS” commands.

   When you send an “RETR” command to the server, the server sends back a two-part response. First it sends a single line telling you whether your command was valid (“+OK” if the message number was acceptable, but “-ERR” if it was not). You should use nextLine to retrieve this line and then display it in your program’s lower text area just as you did for the server’s responses to the “USER” and “PASS” commands.

   If the “RETR” command was valid, the server will next send you all of the lines of the requested email message. There is a special method included in NetConnection that enables you to retrieve all of the lines in the message in one step. The method is named nextPOPResponse. It retrieves all lines sent from the POP server up to the line containing a single period that marks the end of the message. Use this method just as you used
nextLine in your previous commands to retrieve the contents of the email message and display this text in your program’s upper text area.

6. **Validate the RETR Command:** At this point, you should have a working email reader! As long as all of the information the program’s user enters is correct, your program should work as desired. Unfortunately, if the user types in a bad password or a message number that is out of range, things do not work well at all. Try it. Just run your program and press the “Get Message” button without typing anything into the program’s text fields. The “Get Message” button will become permanently highlighted and your program will not respond to any form of input.

The problem is that your program has issued a `nextPOPResponse` request when no POP response is actually available. When you invoke the `nextPOPResponse` method, the computer reads lines from the server until it finds a line containing just a period by itself. Unfortunately, since the server never received a valid “RETR” command, it will never send a period on a line by itself to your program. Your program, however, will obediently follow your instruction and wait forever for the line containing a period to arrive.

To rescue your program from this state, click on the program project window to bring it to the front, point the mouse at the progress bar below the “Compile” button, hold down the control button and depress the mouse, and finally select the “Reset Java Virtual Machine” item that appears.

To prevent this ugly situation, you should revise your code so that the line that uses the `nextPOPResponse` method is only executed if the server’s response to the “RETR” command is “+OK”. Doing this requires a feature of Java that we have not yet discussed in class, the `if` statement. Later in this handout we provide enough information about `if` statements so that you can write the required code. So add the required `if` statement as described in the section “Using a Conditional Statement.”

7. **Check Everything Over:** 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 enter invalid message numbers. In addition, take a few extra minutes to look over your code for any errors that might still exist but did not 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 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. At a minimum, **select the “Auto-layout” item from the BlueJ editor’s “Edit” menu.** This will cause BlueJ to do its best to reformat your code nicely.

   Make sure the names of the variables used in your program help clarify the functions of those variables. Avoid short, cryptic names. Include `final` instance variable declarations to associate names with the values used to specify the dimensions of program text areas and the port number to which you connect. Use these names 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. *In general, commenting groups of related instructions produces code that is more readable than code in which every single line has its own comment (or code in which no comments are provided).* Make sure to include a comment before your class header that includes your name and lab section. Figures 4.5, 4.6 and 4.8 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.

8. Visit the “Labs” page of the course website (http://www.cs.williams.edu/~cs134) and follow the instructions for submitting your work electronically. You will also find a summary of our criteria for grading labs on that page.

**Using BlueJ to Enter and Run Your Program**

You will use BlueJ to enter and run your program much as you did for Part II of last week’s lab.

Begin by launching the BlueJ application (either press command-space followed by typing its name or just click on its icon in the dock). Once BlueJ is running:

1. Select “New Project” from the BlueJ Project menu and use the dialog box that appears to navigate to the “Documents” folder in your account directory.

2. Name your project. If your name happens to be Sally Floyd, then FloydLab2 would be a great project name. Make sure that your project name **ALWAYS** includes your name and lab number and **NEVER** contains blanks or special characters. Enter this name in the field labeled “File:” at the top of the dialog box. Click “Create”.

3. BlueJ should now display a project window with your project name in its title bar. Click the “New Class...” button in this window.

4. BlueJ will display another dialog box asking you to name your class and to tell it what kind of class it is. Select “GUIManager” for the type of the class. Name your class “EmailReader”. Click “OK”.

5. At this point, an icon with the name you picked for your class will appear in the project window. Double-click on this icon to display a window showing the text of your class. Within this window you can use the mouse and keyboard in the usual ways to edit the text of your program.

Because you told BlueJ that your class would be a GUIManager extension, its text will initially consist of a template with skeletal definitions of the constructor, a buttonClicked method, and other methods. You can now begin the first step of our implementation plan by placing the code to create the desired GUI components in the constructor and adding declarations for any needed instance variables. You should also delete the templates for methods that you will not be using (everything but the constructor and buttonClicked method).

When you are ready to test your program, compile and run it by control-clicking and creating a new instance. You should now be able to interact with your program. Try entering a user id or pressing the button. When you are done, select “Quit” from the “BlueJ Virtual Machine” menu to terminate your program. **DO NOT** just close the window by clicking on the red button in the upper-left corner. Actually quit instead.

**Get Wired!**

As explained on the course website, you can download BlueJ and copies of the Squint library if you want to work on any of our lab assignments outside of lab. For this lab and next week’s lab, there is one other step you have to take. You will need to plug your computer into an Ethernet jack. The campus wireless network is configured to reject unencrypted POP packets as a security precaution. As a result, the POP clients you write will not work on Purple Air.

**What if my program doesn’t work?**

While you are working on this lab, a disturbing thing may happen. Your program may not do what you expect. For example, instead of seeing nice “+OK” messages in your text area it may be filled with “-ERR” messages. There is a technique that may help you identify the source of your problem. Before running your own program, start TCPCapture and configure it to capture all POP packets. Then run your program and look at all the packets that are sent between your client and the server. You will see the actual contents of the messages your program sends along with the
“+OK” and “-ERR” message from the server. This will probably provide you with a better understanding of what is going wrong and enable you to fix the problem.

**Summary of POP**

You should be familiar with the operation of the POP protocol from last week’s lab. Just in case, however, we adapted the following fragment from last week’s handout to provide a refresher.

POP operates on port 110. So, when you create a `NetConnection` in your program it should be connected to port 110 on our mail server, `fuji.cs.williams.edu`. Each POP command should be sent as a separate line.

The first two POP commands you will need identify the account whose mail is to be accessed:

**USER** - The first packet sent to the server should be composed of the code “USER” followed by a space and then a user name (such as `19jrl2`).

**PASS** - The second packet sent to the server should be composed of the word “PASS” followed by a space and then the account password.

Next, you will send one command to retrieve the requested message:

**RETR** - The retrieve request must include the number of the message to retrieve.

When you are all done you will send the server a simple “QUIT” command and then close the connection.

The POP server will send a packet back to your client in response to your initial connection and each command it receives. These packets will begin with a “+OK” or “-ERR” code indicating whether the request from your client was acceptable or not. POP servers usually include explanatory information after the “+OK” or “-ERR” summary code.

**Using a JPasswordField**

In this lab, you will use a `JPasswordField`. This is a text field specially designed for inputting passwords. Unlike a normal text field, a `JPasswordField` doesn’t show the characters a user types into the field. Instead it displays a circle or star for each character typed.

While a `JPasswordField` provides a slightly different interface to the user of a program, it can behave just like a `JTextField` from the programmer’s point of view. To make Java aware that you want to treat the `JPasswordField` just like a `JTextField` within your code, you simply declare the variable in your program that will refer to the `JPasswordField` as a `JTextField`. That is, you should use a declaration of the form:

```java
private JTextField password;
```

Even though this variable is declared to refer to a `JTextField`, you will initialize it in your constructor by associating it with a `JPasswordField` using an assignment like:

```java
password = new JPasswordField( FIELD_WIDTH );
```

This is a bit unusual. In examples we have seen previously, the type provided before a variable’s name in its declaration has been the same as the name used with “new” to create the object. Java recognizes, however, that certain types of objects are just special cases of other types. This is very common in the real world. Just as a cat is an animal or milk is a beverage, in Java, a `JPasswordField` is a `JTextField`. 
Using a Conditional Statement

There is a construct in Java that makes it easy to revise your code so that it will not attempt to retrieve the lines of an email message if the server sends a "-ERR" in response to a "RETR" command. It is called an if statement. It allows you to tell Java to choose between executing two sets of instructions based on a specified condition. For example, in your program you can use an if statement to tell Java to choose between using nextPOPResponse to retrieve the lines of an email message or using append to display a message warning the user that the desired message could not be accessed.

An if statement takes the form:

```java
if ( some-condition ) {
    statement(s) to execute if the condition is true
} else {
    statement(s) to execute if the condition is false
}
```

For your program, the condition that should determine which text area to use is whether the latest line received from the server begins with "+OK" or not. If the name lineFromServer refers to the latest line retrieved from the server, you can express this choice in Java by saying

```java
if ( lineFromServer.startsWith( "+OK" ) ) {
    statement(s) to execute if condition is true
} else {
    statement(s) to execute if condition is false
}
```

The statement(s) included within the first pair of curly braces should invoke nextPOPResponse and append the text retrieved to your program’s upper text area. The statements included within the second pair of curly braces should display an error message in the upper text area. In order to use the variable lineFromServer in the if statement, you will need to declare lineFromServer as a String variable, get the next line from the server, and associate this line with the variable using an assignment statement before the if statement is executed.

Modify your POP program to include such an if statement.