Lab 2

A Minimal Email Reader

Due: Sept. 23/24, 11 PM

This week's lab assignment is to write a simple email client program. The goals of the lab are:

- to reinforce your understanding of the POP protocol, and
- to write your first Java program.

The program you write will actually only provide half of the minimal functionality of an email client. It will enable its user to read mail but not provide the ability to send mail.

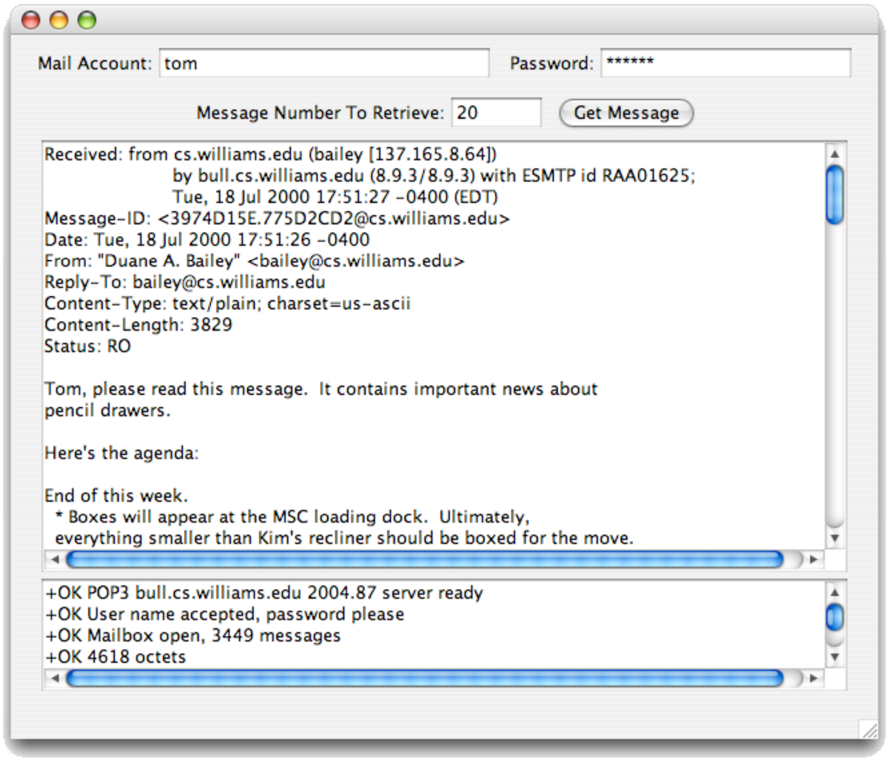
This handout provides guidance on the steps you will need to complete as you construct this program. **You should read the entire handout carefully <u>before</u> coming to lab.** In addition, you should sketch out the code you think you will need for at least the first steps of the lab on paper before your lab period.

In this handout, we first describe the interface that your program should provide. We then discuss the major steps you will have to take to complete the program. Next, we review the details of the process of entering your program's code, running your program, and electronically submitting your work when you are done. After that, we explain what we will look for when we grade it. Finally, we provide information on a number of details you will need to complete this program including a short review of the POP protocol.

A POP Client Interface

A sample of what the interface for your program might look like is shown below. Across the top of its window, the program displays two text fields in which the user must enter his or her account information. Below these is 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 program's window is filled with two `JTextAreas`. The upper text area is used to display the latest message retrieved. The text area that appears near the bottom of the window is used to display the "+OK" and "-ERR" messages the POP server sends in response to the commands it receives from your program.

While experimenting with the Apple Mail program, we learned that it fetches all available messages at once. Your program will take a different approach. Each time the "Get Message" button is pressed, it 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. As a consequence, the user of the program must fill in all three text fields before pressing the "Get Message" button.



Mail Account: tom   Password: ******

Message Number To Retrieve: 20

```
Received: from cs.williams.edu (bailey [137.165.8.64])
                by bull.cs.williams.edu (8.9.3/8.9.3) with ESMTP id RAA01625;
                Tue, 18 Jul 2000 17:51:27 -0400 (EDT)
Message-ID: <3974D15E.775D2CD2@cs.williams.edu>
Date: Tue, 18 Jul 2000 17:51:26 -0400
From: "Duane A. Bailey" <bailey@cs.williams.edu>
Reply-To: bailey@cs.williams.edu
Content-Type: text/plain; charset=us-ascii
Content-Length: 3829
Status: RO

Tom, please read this message.  It contains important news about
pencil drawers.

Here's the agenda:

End of this week.
 * Boxes will appear at the MSC loading dock.  Ultimately,
 everything smaller than Kim's recliner should be boxed for the move.
```

```
+OK POP3 bull.cs.williams.edu 2004.87 server ready
+OK User name accepted, password please
+OK Mailbox open, 3449 messages
+OK 4618 octets
```

Implementation plan

It is a good idea to implement a program in small steps, testing the correctness of what you have done in 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. 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.

   Test that the code you write works by running the program (as described later in this handout) to verify that the interface appears as desired. In fact, you will probably want to run the program as soon as you have written enough instructions to create the first few elements of the interface and repeatedly as you add more components.

2. 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. As a result, in addition to creating the connection you should write code to retrieve and display this welcome message in your program's upper text area.

   a. Declare an instance variable to refer to your `NetConnection`, but place the assignment statement that actually constructs the connection and associates it with the variable name in the `buttonClicked` method.

   b. After the code that creates the `NetConnection`, place an instruction to add the program as a message listener for the connection. Doing this will indicate to the program that you plan to handle all messages received from the server within a separate `dataAvailable` method. That is, your final code is likely to look more like the code in Figure 4.11 of "Programming with Java, Swing, and Squint" than like Figure 4.8.

   c. Define a `dataAvailable` method. The instructions in this method will be executed each time a message arrives from the server. Place code in this method to retrieve a line from the `NetConnection` and display it in the program's larger text area. Don't forget to add a new line character (\n) to the end of each line.

   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.

   Note that when you run your program, the "+OK" from the server appears in the text area at the top of your window. This is not where you will ultimately want this line to appear. The content's of the email message retrieved should be displayed in the upper text area, while the "+OK" messages should appear in the lower text area. For now, however, leave your code as it is. We will correct the placement of these messages in step 6.

3. Next, actually send a command to the server. Start with the simplest command, "QUIT". This should cause the server to send back another "+OK" line. Run your program and make sure that this extra line appears.

4. Programs that use `NetConnections` should close the connections once they are no longer needed. Your program does not currently do this. Luckily, when you send a "QUIT" command the server will close its end of the connection. Therefore, you can easily arrange to close your end of the `NetConnection` by defining a method named `connectionClosed`. The code in this method will be executed as soon as the server closes the connection. Place code to close your end of the `NetConnection` in this method's body.

   To verify that the connection is actually being closed, include code in your `connectionClosed` method that will append a message to your text area indicating that the connection has been closed.

5. Now that the connection is being constructed and closed correctly, you can write instructions to send the commands required to login and to fetch a message. Place the code to send these commands in your buttonClicked method before the code that sends the "QUIT" command.

6. As explained above, the upper text area in your program's window is supposed to hold the text of email messages while the lower text area should hold the "+OK" responses from the server. We postponed implementing this behavior because it 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 to enable you to write the required code. Our hope is that this limited, but practical experience using an `if` statement in lab will enable you to better understand our presentation of the details of using if statements in class.

   So, as a final step in your implementation, add the required if statement as described in the section "Using a Conditional Statement."

7. 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 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. Make sure the names you chose for the variables used in your program help clarify the functions of those variables. 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, 4.6 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.

Using BlueJ to Enter and Run Your Program

You will use BlueJ to enter and run your program much as you did at the end of last week's lab.

Begin by launching the BlueJ application (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 folder name **ALWAYS** includes your name and **NEVER** contains blanks or special characters. Enter this name in the field labeled "File:" at the top of the dialog box.
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`".
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 displaying 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 that includes 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.

When you are ready to test your program, you can tell BlueJ to execute your code as follows:

1. Press the "Compile" button to tell BlueJ to check your code for syntactic errors and convert it into its internal form, Java virtual machine code.
2. Point the mouse at the icon for your class in the project window, depress the "ctrl" (control) key, and then depress the mouse button to make a menu appear.
3. Select the first item in the menu. It should look like "`new EmailReader()`". This tells BlueJ to create an instance of your class and to execute the instructions you placed in its constructor. Just use the default name for

your instance provided by BlueJ. An icon for this new instance will appear in the bottom of the project window. A window will appear on the screen and any GUI components your code creates will appear within the window.

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.

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 should provide you with a better understanding of what is going wrong and enable you to fix the problem.

Submission Instructions

Return to the Finder and look in your "Documents" folder. You should find the folder that BlueJ created for your project. Its name should be the one you picked for your project (something like `FloydLab2`).

- 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 `FloydLab2`) 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.

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 submit and later discover that your submission was flawed, you can submit again. We will grade the latest submission made before the 11 p.m. deadline. 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 "re-vised") and the re-submission should work fine.

Reality Check!

Admittedly, the program you have just completed provides far less functionality than any real mail client. It is, however, *real* in the sense that the protocol it uses is the same protocol used by real mail servers. This means that, you should be able to use it to read mail in any of your real mail accounts. If you are curious, you can verify this by making a few small changes.

If you use your Williams email account, you can read the mail stored there by just making one simple change to your program. Instead of connecting to the server `cortland.cs.williams.edu`, change your program to connect to `studentmail.williams.edu`. Then, use your OIT account ID and password when contacting the server.

If you use Gmail, you will have to do a little more, but not much. Again, you will use a different server. The server's name is `pop.gmail.com`. The other changes required result from the fact that Google Mail uses a different type of connection to provide more privacy/security. First, instead of constructing a new `NetConnection` in your program, you should create a new `SecureNetConnection`. From your program's point of view, a `SecureNet-Connection` works just like a `NetConnection`, but this new class will send your data in an encrypted form. You will also need to change the port number you connect to from 110 to 995. Once you make these changes, run your program, enter your Gmail account ID and password, and you should be able to retrieve messages.

Grading

Grading will typically be done by evaluating your lab submission based on three sets of criteria: how complete your code is, how well it works, and how clearly it is written. Completeness measures whether you have done a reasonable job of writing instructions to provide the desired functionality, even if the instructions you wrote don't fully work. Correctness measures whether all the desired functionality works as we expect it to. Good style is also an important quality for programs to have; it is essential to make programs readable and understandable. Over the course of the semester we will get pickier about style issues, but it is important for you to get into good habits from the very beginning.

Completeness / Correctness

- GUI layout
- Connecting to and disconnecting from the server
- Logging in
- Retrieving mail messages
- Displaying mail messages in top text area
- Displaying server responses in bottom text area

Style

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

Programming labs will be graded on the following scale:

++	An absolutely fantastic submission of the sort that will only come along a few times during the semester.
+	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 submission that satisfies all the requirements for the assignment --- a job well done.
✓	A submission that meets the requirements for the assignment, possibly with a few small problems.
✓-	A submission that has problems serious enough to fall short of the requirements for the assignment.
-	A submission that is significantly incomplete, but nonetheless shows some effort and understanding.
--	A submission that shows little effort and does not represent passing work.

Using a JPasswordField

In this lab, you should use a form of GUI component called a `JPasswordField`. This is a text field specially designed for inputing passwords. Unlike a normal text field, a `JPasswordField` doesn't show the characters a user types into the field. Instead it displays a 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, instead of using a declaration like

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

you should use a declaration of the form

```
    JTextField 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 associated with the variable. 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 fairly easy to revise the `dataAvailable` method in such a way that the "+OK" responses from the server will not appear in the same text area as the contents of email message you retrieve. It is called an `if` statement. It allows you to tell Java to choose between executing two sets of instructions in a method based on a specified condition. For example, in your mail program you could use an `if` statement to enable Java to choose whether a line retrieved within your `dataAvailable` method should be displayed in the top text area or the bottom text area based on whether or not the line began with the text "+OK".

An `if` statement takes the form:

```
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

```
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 append the contents of the line received to the lower text area. Statements to display the line received in the upper text area should be placed between the second pair of curly braces. In order to use this code, you will have to place lines to declare the variable `linesFromServer`, to get the next line from the server, and to associate this line with the variable `linesFromServer` before the `if` statement.

We want you to modify your POP program so that the final version of your `dataAvailable` method includes such an `if` statement. Of course, this code will be fooled by any email message that contains a line that actually starts with "+OK" or if the server sends a "-ERR" response. We will worry about how to deal with that issue next week....

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, cortland.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 jrl2).

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 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 includes explanatory information after the “+OK” or “-ERR” summary code.