Lab 4

What’s on the Menu

Due: Wed. 2/28 at 11PM (for Mon. aft. lab), Thurs. 3/1 at 5PM (for Mon. evening), or Thurs. 3/1 at 11 (for Tues. aft.)

The goal of this week’s lab is to employ \texttt{String} methods and loops to produce a new email reading program with a more convenient interface.

The type of window that should be displayed by the program you construct this week is shown below.

At the top of the window, there are text fields to allow a user to enter an account identifier and a password. There is also a single button. Initially, the button will display the label “Log in”. When it is pressed, the program will attempt to connect to the POP server on fuji.cs.williams.edu and log in using the account information.

If the program is able to log in to the POP server, it changes the label of the button to “Log out”. At the same time, it contacts the POP server to download the headers of all the emails stored on the account. From the header lines of each message it extracts a short description of each message and builds a menu that appears at the bottom of the screen.

When the menu is first filled and whenever the user chooses a new item from the menu, the program downloads the contents of the message corresponding to the menu item currently selected and displays this message in a text area that fills the middle of the program window. The information displayed is not identical to what was received from the POP server. In particular, the program only displays the “interesting” parts of the message’s header. In addition, if the message received was encoded using the multi-part MIME format, the program only displays the “text/plain” subsection of the MIME content.
Getting started
In lab 1, we had you send yourself some deliberately peculiar email messages. In the last two weeks, many of you have added even more silly messages. To make sure you have a nicer collection of emails to work with this week, we would like you to a) delete all the messages currently in your account, and then b) send some “imaginative” messages to several (8 actually!) of the other students in your lab section. The tricky part is to make sure that you all finish the deleting part before any of you start sending the improved messages.

So, before you do anything else in lab, let’s delete your existing messages. The most reliable way to do this is with NetTap.

- Use NetTap to connect to fuji.cs.williams.edu on the POP port (110).
- Send USER and PASS commands using your CS account and password.
- Send a STAT command to see how many messages you have.
- Send a DELE command for each message from 1 up to the value reported by STAT. That is, send commands of the form “DELE 1”, “DELE 2”, …
- Finally (AND THIS IS ESSENTIAL!), send a QUIT command to the server.

Now, let’s do something else before sending any messages to make sure everyone gets their old messages deleted.

Even though this program is clearly closely related to last week’s assignment, we recommend that you start over with a completely new BlueJ project this week. The structure of the new program is sufficiently different from last week’s program that we believe you will be less confused and end up with better final code if you do this. So, to start, launch BlueJ, create a new project in your Documents folder with a name like jrl3-Lab4 composed of your OIT login followed by the name of this lab. Then, create a new class. Just to make sure that it is distinguishable from the last two labs, we suggest you call this class “Tweety-Bird” in honor of the Thunderbird mail client you used in lab 1.

Now, share your CS login id with 8 other students in the class and collect 8 email addresses from other students in the class. Once you have these 8 addresses, send a creative message (at least make it a few lines long) to all 8 of the other students. Note, you don’t have to send 8 messages. Just send one message to eight addresses. All of the addresses should end with “fuji.cs.williams.edu”. Different mail programs send messages differently and variety would be good here. So, if some of you use the Gmail web interface while others use apps on your phones or even Thunderbird on one of our Macs we will have better variety to help debug the code you write this week.

Implementation plan
1. **Build the User Interface:** Start by just typing in the instance variable declarations for all of the user interface elements and the code needed in the constructor to create the desired interface. If you look at our screen shot of the program that we believe you will be less confused and end up with better final code if you do this. So, to start, launch BlueJ, create a new project in your Documents folder with a name like jrl3-Lab4 composed of your OIT login followed by the name of this lab. Then, create a new class. Just to make sure that it is distinguishable from the last two labs, we suggest you call this class “Tweety-Bird” in honor of the Thunderbird mail client you used in lab 1.

Now, share your CS login id with 8 other students in the class and collect 8 email addresses from other students in the class. Once you have these 8 addresses, send a creative message (at least make it a few lines long) to all 8 of the other students. Note, you don’t have to send 8 messages. Just send one message to eight addresses. All of the addresses should end with “fuji.cs.williams.edu”. Different mail programs send messages differently and variety would be good here. So, if some of you use the Gmail web interface while others use apps on your phones or even Thunderbird on one of our Macs we will have better variety to help debug the code you write this week.

2. **Handle logging in and out:** Define an initial version of the buttonClicked method that will log in to (and out of) the POP server.

The behavior of this method will not depend on which button was clicked. This program will only have one button! Instead, it will depend on whether or not the program is currently logged in to the POP server or not.
To keep track of whether the program is logged in or not, define a boolean variable with a name like “connected” or “loggedIn”. This variable should initially be set to false. When buttonClicked begins executing, it should check to see whether this variable is true or false using an if statement. Since the only requirement on the condition you provide in an if statement is that it describes a boolean value, the most concise and best way to write such an if statement is to use the variable or its negation as in

\[
\text{if ( connected ) } \{ \ldots \}
\]

or

\[
\text{if ( ! loggedIn ) } \{ \ldots \}
\]

In particular, you should avoid comparing the value of the variable to true or false as in “if ( connected == true ) {…}”. Doing this is redundant (and can lead to subtle errors).

If the variable you use is false, then the buttonClicked method should send USER and PASS commands to the server with the user’s login information. If the server responds with “+OK”, then the variable should be set to true and the text of the program’s button should be changed to “Log out”. If the variable is already true when buttonClicked is executed, the method should send a QUIT to the server, close the connection, set the text of the button back to “Log in” and finally set the variable that keeps track of whether the program is connected back to false.

3. Build a simple menu of message numbers: Next, add code to buttonClicked to fill in the JComboBox at the bottom of your program’s window. We don’t, however, want you to try to fill the menu in with short summaries of the messages available at this point. Instead, you should just fill it in with the numbers of the messages available in the user’s account as shown below.

To do this, you should use the POP STAT command, and a simple counting loop.

The STAT command causes POP to send the client program a message of the form

\[+OK 18 \ 23920\]

where the number in the middle (18 in this case) is a count of the number of messages available in the user’s mailbox. Your program should use this command to retrieve such a String from the server and
then use the `indexOf` and `substring` methods to extract the message count. You can use a method named `Integer.parseInt` to convert the `String` version of this number into an `int` value. This method is discussed in a later section. Finally, write a loop based on this `int` value to insert the numbers from 1 to the number of messages into the menu.

Now that your menu is filled, remember to modify the code that handles logging out to empty it. This can be done using a `JComboBox` method named `removeAllItems`.

4. **Fetch and display full message contents:** At this point, your program should have a `JComboBox` full of message numbers, but no `menuItemSelected` method to actually react when a message’s number is selected. You should now add a simple version of `menuItemSelected`. The initial version of this method should use the `selectedIndex` method to access the position of the menu item selected. The positions of menu items in a `JComboBox` are counted starting with position 0 (just like letters within a `String`). Therefore, you will have to add 1 to the position of the selected menu item to compute the number of the associated email message. Send this number to the POP server. Then, retrieve the email message from the server and display its full contents in the program’s `JTextArea`.

Note: Last week we made you fetch the message line by line using the `nextLine` method and a loop and we made you skip the uninteresting header lines. At this point, you should make your life easy and just use `nextPOPResponse` to fetch the entire message (as you did in Lab 2) and display everything (including all the header lines). We will guide you to filter the messy parts later.

The thing you will need to be careful about in this step is the tendency that Java has to execute the code in the `menuItemSelected` method in some cases when no menu item is actually selected. Sometimes, when you first create a menu but have not even added any items, `menuItemSelected` will execute and in this situation `selectedIndex` will return -1. The simplest defense it to check that the value returned by `selectedIndex` is a positive number. The other step you should take is to use `setEnabled` in your constructor and in the `buttonClicked` method to make sure that the `JComboBox` is enabled only when it is filled with menu items.

5. **Fill your menu with message summaries:** Your `buttonClicked` method already contains a loop to fill each entry of your `JComboBox` with the number of an email message. Now, you should add code to this loop to fill each entry in the `JComboBox` with a combination of the message’s number and a summary of its contents.

You can (and should!) do this without retrieving all of the content of all of the messages in your account. POP provides a command named TOP. TOP is described in more detail in a subsection of this handout following the implementation plan. As in the preceding step, after your program sends the TOP command it should use the `nextPOPResponse` method to retrieve all the lines the server sends back in a single step. Then, you should use `String` methods to extract the from address and the subject line from the headers and combine them to form an item to add to your `JComboBox`. The format we used is fairly simple. Each menu item contains the message number, followed by the from address and ends with the message subject. You are free to be more creative.

Since this will be your first real experience using `indexOf` and `substring`, we suggest you proceed one step at a time. First, just extract the subject line from the headers and fill each menu item with just the subject. Then, figure out how to extract the from address (the code should be very similar to the subject line code). Finally, add the message number. Make sure that your code works even if the message in question has no subject header!

6. **Filter the header line:** You should now start working on your `menuItemSelected` method to improve how it actually displays a message when a menu item is selected. Eventually, we want you to filter out just the most interesting parts of both the message headers and the message body. For now, focus on the headers.
The work we want you to do for this step is essentially the same as the last step you performed for Lab 3. Right now, your `menuItemSelected` sends a RETR command to the server and then uses a single `nextPOPResponse` to retrieve the entire contents. Instead, after sending the RETR command, we want you to execute a loop that will retrieve the header lines one-by-one while only appending the interesting ones to the contents of the `JTextArea`. After this loop reaches the end of the headers (an empty line), you should next use `nextPOPResponse` to retrieve the rest of what the server sent and append it to the `JTextArea` after the headers.

7. **Find the MIME boundary string (if possible):** In a multi-part message using MIME, the subparts of the message are separated from one another by lines containing a very unique looking boundary string. The exact contents of this boundary string are provided in a header line that might look like:

   Content-Type: multipart/related; boundary="94eb2c0b200004d39d0565d0fb4e"

   In this example, the boundary string is 94eb2c0b200004d39d0565d0fb4e.

   If you wanted to be very careful to find the right boundary value, you would first make sure that you found a header line that started with “Content-Type:” and then search for “boundary=” within that line. For this lab, we want you to take a little shortcut. As the loop you wrote for the preceding steps makes its way through all the header lines, it should check each line to see if it contains “boundary=”. If it finds such a line, it should save the string between the quotes after “boundary=” in a local variable. This variable should be initially set to the empty string before the loop. In this way, when your header loop finishes this variable will either hold the boundary string if the message uses MIME or be an empty string if not.

   You will use this boundary string in the final step. For now, you should just display it in your `JTextArea` after the headers so that you have evidence that your code is extracting it correctly.

8. **Display only the text/plain part of multi-part MIME messages:** The final step is to write a loop using `indexOf` and `substring` to search through the subparts of a multi-part MIME message looking for the sub-part identified as “text/plain” and then to display only the contents of this subpart in your `JTextArea`.

   This loop should only be applied if the message is actually encoded using MIME. So, your code should first test whether a boundary line was found in the header section of the message. If not, the entire message body should be appended to your `JTextArea`. If the boundary string was found, then you should only display the “text/plain” subpart.

   You will be using a loop here, but it will not be a loop that reads the text sent by the server line-by-line as you did to process the headers. Instead, you will still use `nextPOPResponse` to read everything the server sent you after the headers. You will then follow the statement that invokes `nextPOPResponse` with a loop that uses `String` methods to identify the sub-parts of the `String` returned by `nextPOPResponse`.

   As explained below, in a MIME multi-part message, each subpart is further sub-divided into a header section and a body section by a blank line. Your loop should use the presence of copies of the boundary string your code found in the headers to extract each subpart. Then, you should use the blank line that separates each subsections headers and body to extract the headers from the sub-part. If these headers do not include a string of the form “Content-Type: text/plain” you should skip the subpart. If the subpart’s header does identify it as text/plain, then you should append the body of the subpart to your `JTextArea`.
9. 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.

**Using BorderLayoutManager**

In our first labs, we have taken a very simple approach to the positioning of GUI components within our program windows. Basically, as we have added components to the `contentPane`, they have simply filled the window from left to right until the width of the program window is filled. Then, the program starts a new row of components below the earlier row until the new row again fills the window. This approach is called a flow layout and is the default for a `GUIManager`’s `contentPane`.

The Java GUI libraries provide a rich collection of other approaches to controlling the layout of GUI components. With this program, we want to introduce an approach called `BorderLayout`. When a program’s window is controlled with a `BorderLayout` manager, the `contentPane` is divided into five sub-regions cleverly named `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.WEST`, `BorderLayout.EAST` and `BorderLayout.CENTER`. In the program image shown on the first page of this handout, we are using the `NORTH`, `CENTER` and `SOUTH` regions. The `NORTH` holds the login components, the `CENTER` holds our `JTextArea`, and the `SOUTH` holds the message menu.

To use a `BorderLayout`, we must apply a method named `setLayout` to the `contentPane` before adding any components. This method expects us to pass the desired layout manager as a parameter. In particular, to use a `BorderLayout`, we would include the line

```java
contentPane.setLayout( new BorderLayout() );
```

among the first lines of our program’s constructor.

The class `BorderLayout` is part of the standard Java libraries, but rather than belonging to Swing, it belongs to an older library known as the Abstract Windowing Toolkit or AWT for short. As a result, to make it possible to use `BorderLayout`, one must add a line of the form

```java
import java.awt.*;
```

at the very start of the program file (with the other import commands before the class header).

Once you have told your `contentPane` to use a `BorderLayout`, you must provide information about where to place each component you add to the `contentPane`. This is done by providing the name of one of the five subregions as a second parameter when you say `contentPane.add`. For example, the command to add our `JComboBox` to the `contentPane` looked like

```java
contentPane.add( messageMenu, BorderLayout.SOUTH );
```

Each of the five regions associated with a `BorderLayout` can only hold a single component. This may seem to contradict that fact that our program’s window appears to have five components in the `NORTH` region. This contradiction is resolved by using `JPanel`s. To put multiple components in any region, we create a `JPanel`, add all the desired components to the `JPanel` and then add the `JPanel` to the desired region of the `contentPane`. In this case, the commands that add components to the `JPanel` will not be given second parameters specifying subregions. The elements added to the `JPanel` will simply be positioned left to right as they have been in past labs. When we add the `JPanel` to the `contentPane`, however, we need to specify the subregion as a second parameter.
One advantage of using a BorderLayout manager is that it stretches the components we place in our window so that they fill the window. In particular, components added to the NORTH or SOUTH are stretched to use the full width of the window and whatever is added to the CENTER is stretched to fill all the space not otherwise used by the other four sub-regions. As a result, you will discover that as you resize your program’s window this week, the menu at the JTextArea grow and shrink with the window.

**Integer.parseInt**
There are several ways to represent a numeric value in Java. If we type “37” Java views the value as text that happens to only contain digits but still sees this as a member of the String type. On the other hand, if we type 37 without quotes Java understands that this value is a member of the type int. If we apply the + operator to two String values, it just sticks them together. Thus, “37” + “1” would evaluate to “371” while 37 + 1 produces 38 as expected. As a result, it is critical in some situations to explicitly tell Java to convert a String value composed only of digits into an int value.

This can be accomplished using a library method named Integer.parseInt. In this program, for example, you will use the POP STAT command to obtain a String of the form “+OK 37 2341” where the 37 in the middle of the String is the number of messages currently stored on the account. You can extract the two digits 3 and 7 using a command like

```java
String digits = lineFromServer.substring( 4, 6 );
```

The variable digits, however, is not an int value. To obtain the desired int value, you would take the extra step of saying

```java
int messageCount = Integer.parseInt( digits );
```

Be warned that the Integer.parseInt method is very picky. The String to which it is applied must only contain digits. If the String contains even an extra space or a comma, a program error will result.

**The POP TOP command**
The POP protocol includes a command that makes it possible to access the headers and the first few lines of a message’s body without retrieving the entire message body. This command is intended to allow a mail client program to efficiently construct summaries of the messages in a user’s mailbox without transmitting the full contents of all of the message in the user’s mailbox through the network.

The command is named TOP. It expects two arguments. The first argument is a message number much like one would provide to the RETR command. The second argument specifies the number of lines beyond the headers that should be provided. Like the RETR command, the server will send a single line containing just a period after it sends the last line requested by a TOP command.

As an example, the command

```
TOP 11 0
```

could be used to request just the header lines of message number 11.

**MIME message structure**
In the last two labs, many of you have noticed that some of the emails your programs have displayed are difficult to read because the text of the messages are embedded in a collection of confusing header lines. For example, if you viewed the message shown on the front page of this handout with a program that did not remove these confusing headers it might look like the window shown on the next page.
If you look carefully, you will find the expected text of the message fills roughly the second quarter of the screen vertically. It is, however, preceded by several unexpected headers and followed by text that contains the words of the message surrounded with mysterious symbols.

This message is an example of multi-part MIME encoding. There are several key features you must recognize to understand the encoding being used.

In the fifth line of the message, we see a header that describes the “Content Type” of the message as multipart/alternative. This line contains the strange string 001a11402d968564750565d0918e which is identified as the “boundary”. If you scan through the rest of the message, you will discover that this boundary appears three times later in the message preceded by “--”. These boundaries are intended to divide this message up into two subparts. Each subpart of the message is preceded and followed by a line containing the boundary sequence.

Each subpart of the message also has a substructure that mimics that of a complete email message. Each subpart is divided into an initial set of header lines that is separated from the body of the subpart by a single empty line. The first subpart has a single “Content-Type” header immediately followed by a blank line and then a 9 line long body. The second subpart has two header lines followed by a blank line and a 6 line body.

Note that one of the headers in each subpart starts with “Content-Type:”. This header line provides the information to identify the role of each subpart. Almost all multipart MIME message will include a subpart whose “Content-Type” header says “text/plain”. This is the simplest version of the message being
transmitted. In particular, the body of the “text/plain” subpart of an email message is what your program should display in its JTextArea.

There are many other possible types of MIME subparts. When you attach a file to an email, it will appear as a subpart. In the example, the mailer has included a second version of the original message encoded with HTML, the language used to describe web pages. It is very common for modern mail clients to do this. It makes it possible to transmit messages with changes in font size, text color, etc. that are not possible with plain text.

**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 optional features. 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. You have seen that many email message arrive with a version of the message encoded using HTML as one of the MIME subparts. We had you ignore this subpart and use the plain text version. An alternative is to use the HTML subpart of each message and a Java GUI component that knows how to display HTML tags. The Java JEditorPane is such a component. This will be easy to do if you already know HTML. If not, feel free to ask for help.

   To use a JEditorPane you will need to:

   a) Change the variable you used to refer to your JTextArea to a JEditorPane variable and construct a JEditorPane instead of a JTextArea (the constructor expects no parameters). Assuming the JEditorPane variable’s name is conversation, include the invocations

   ```java
   conversation.setEditable( false );
   conversation.setContentType( "text/html" );
   ```

   in your constructor (the first one should really be there already).

   b) Use setText instead of append to place text in the JEditorPane. This means you will have to keep a separate String variable containing the contents of the entire message and its headers. Make sure that the argument to setText begins with `<html><body>`, ends with `</body></html>`, and includes either `<br>` or `<p>` tags between header lines. You can add any other HTML tags you want to make the text appear in color, bold font, etc.

2. Most real email accounts hold more than 8 messages. As a result, most real email programs provide a way to search to find all messages from a particular person or all messages that contain a particular word in the subject. In real mail programs, a search like this usually reduces the “menu” of message
headers available to those that match the search. In this program, it is probably best not to remove items from the menu. You could however provide a form of search which causes the program to select the next item in your message menu that matches the search term each time a button is clicked. Thus, by repeatedly clicking the search button you could easily get your program to help you find all the messages from a particular person or about a particular subject.

For this extension it will probably help to know that the JComboBox class includes two methods named getItemCount and getItemAt.

3. I will try to think of more possible extensions over the weekend!

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. Starting this week, 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 and 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 at

http://www.cs.williams.edu/~cs134/Labs.html