In this week’s lab we will continue to work on the email client you started last week. A sample of what your completed program’s interface might look like is shown below.

If you only look at this image quickly, you might think it is identical to the interface supported by last week’s program. If you look more closely you will realize that it is in fact identical to the interface from last week. That’s right! The functionality of the program we want you to build this week will be identical to what you did last week.

While this week’s program will behave just like last week’s program, its internal organization will be quite different. Instead of putting all the code in a small number of methods within one class, this week we will organize the code into three separate classes (and many methods). You will end up writing very little new code this week, but you will move code around quite a bit. Your main task will be to reorganize the code you wrote last week into separate methods and classes.

A critical step in the process of implementing a program using several classes is deciding which portions of the information needed by the entire program will be kept in each class. This in turn determines which instance variables and which methods are declared in each class. To simplify the process of writing your first multi-class program, we will provide you with suggestions for the instance variables and methods for each of the classes you should define. These classes and methods will be described in the next section. After that, we will provide an implementation plan to guide you through implementing the pieces.

**Class Structure**
You will define three separate classes:

- **TweetyBird**: This class will create the interface that allows a user to display messages by selecting them from a menu and to login or logout using a button. Although we use the same name for this class that we used for your entire program last week, this week’s TweetyBird will be different from last week’s TweetyBird in
many ways. It will only contain one of the three loops that were present in last week’s class and you will not be
using the String methods indexOf and substring or the NetConnection methods in.nextLine or
out.println anywhere within this class.

• POPConnection: This class will make the connection to the POP server and provide methods that the
TweetyBird class can use to do things like retrieve a message or determine the total number of messages
available. The code to send commands in the appropriate format to the POP server and the code to interpret
the server’s responses to these commands will be found within the constructor and methods of this class.

• MailMessage: Your program will create objects of this class to hold each message it retrieves from the
server. The class will provide methods that return various String values related to the email message including
a short summary of its From and Subject headers to use in the program’s menu, and a String suitable for
displaying the message’s interesting headers and the “plain/text” version of its body.

TweetyBird
Your TweetyBird class will be quite similar to the class that was your entire program last week. The class will
contain all of the instance variables for the GUI components used in last week’s lab. You will still need the text
fields and button used for logging in and out along with the menu to select a message for display and the
JTextArea used to display messages.

At the same time, there will be some significant changes. Your new TweetyBird will no longer have an instance
variable of type NetConnection. Instead, it will use a variable of type POPConnection to interact with the server.
The code that was present in TweetyBird last week to send commands like USER, TOP and RETR through the
NetConnection will be moved from TweetyBird to the constructor and methods of the POPConnection
class. In addition, almost all the code that used String methods like indexOf and substring to extract
subparts of the information sent by the POP server will be moved from TweetyBird to your new MailMessage
class.

The overall goal will be that in the end, the details of the program that involve the form of its user interface will
mainly be found in TweetyBird. The details of the program that depend on the rules of the POP protocol will
mainly be found in POPConnection and the details that relate to the standards for encoding email messages
(particularly MIME), will be in the MailMessage class.

The TweetyBird class will still have definitions of the two methods buttonClicked and
menuItemSelected. The buttonClicked method will still consist of a two-way if statement that either logs in
to or logs out from the POP server depending on whether the program is currently connected or not. The
menuItemSelected method will still need an if statement to check that the index of the selected item is a
positive number, but almost all of the if statements and while loops that were included in menuItemSelected
last week will be moved to methods in the MailMessage class. The new version of menuItemSelected will
accomplish its purpose by invoking these methods of the MailMessage class.

POPConnection
Your POPConnection class should provide convenient methods for interacting with a POP server.

The constructor for a new POPConnection should expect three parameters:

- the full network address of the POP server (i.e., “fuji.cs.williams.edu”),
- the user id of the account containing the mail that will be accessed, and
- the password associated with the user id passed as the second parameter.

This class will declare at least two instance variables. One of these will be of type NetConnection. The
constructor for a POPConnection will create a new NetConnection and associate it with this name. The other
instance variable will be a boolean which will be set to true by the constructor if the attempt to log in is
accepted by the server. You may want to include some final instance variables (one for the port number?), but it
unlikely that you will need any other non-final instance variables.

You will need to define five methods in this class:

• isConnected
  This method will be an accessor method that returns a boolean value. It will take no parameters. It should
return true if the POPConnection currently is successfully logged in to the POP server and false if either the attempt to login failed or the logout method has already been invoked.

- **getMessageCount**
  This method will be an accessor method that returns an int value. It will take no parameters. If the POPConnection is currently logged in to the server, this method should return the number of messages available in the account used to log in. If the POPConnection is not connected, this method should return 0.

- **getMessage**
  This method will be an accessor method that returns a MailMessage as a result. It will expect a single parameter value of type int specifying the number of the message within the account that is being requested. If the POPConnection is currently logged in to the server and the message number is valid (i.e., between 1 and the number of messages available), the MailMessage object returned should contain the complete headers and body for the requested message. Otherwise, the method should return null.

- **getMessageHeaders**
  This method will be an accessor method that returns a MailMessage as a result. It will expect a single parameter value of type int specifying the number of the message within the account that is being requested. If the POPConnection is currently logged in to the server and the message number is valid (i.e., between 1 and the number of messages available), the MailMessage object returned should contain the complete headers for the requested message and an empty body. Otherwise, the method should return null.

- **logout**
  This method will be the only mutator method defined in this class. It will take no parameters. It should terminate the POPConnection's connection to the server. After this method has been used, any future invocation of the isConnected method should return false.

---

**MailMessage**

The MailMessage class is intended to provide convenient access to the types of information about a mail message that the TweetyBird class needs to enable a user to view message contents through its interface.

The constructor for this class should expect a single String value as its parameter. The String value provided as a parameter when constructing a MailMessage will either be the full text of an email message (i.e., it will include all the header and body lines) or the header lines followed by a blank line and then some (possibly empty) portion of the body of the message. Note that in the descriptions of the getMessage and getMessageHeaders methods of the POPConnection we said that both of these methods will return a MailMessage as a result rather than merely returning the String received from the server.

The class should include two String instance variables. The code in the constructor for the class should set one of these variables equal to the headers found in the String provided as a parameter to the constructor and it should set the second variable equal to the (possibly empty) body included in the parameter value. To accomplish this, the code of the constructor should use indexOf to find the blank line separating the headers from the body and the substring method to extract the headers and body.

You will need to define two public methods in this class:

- **getSummary**
  This method will be an accessor method that returns a String value. It will take no parameters. It should return a String consisting of a one line summary of the contents of the message suitable for inclusion as an item in the menu displayed by the TweetyBird class. That is, the String returned should include the contents of the message’s “From:” header line (if such a line exists) followed by the contents of the “Subject:” header line (if one is found).

- **getHeadersAndText**
  This method will be an accessor method that returns a String value. It will take no parameters. The String it returns should be a String appropriate to display in the JTextArea of your TweetyBird class. It should begin with copies of the message’s “From:”, “Subject:” and “Date:” header lines. These should be followed by a blank line. Next, if the message is encoded using the multi-part MIME format, the contents of the body of the “text/plain” subpart should be included. On the other hand, if the message is not a multi-part message (i.e., if no “boundary” specification was included in its headers), then the full contents of the message body should follow the three header lines.
In addition to the `getSummary` and `getHeadersAndText` methods which will be used by the `TweetyBird` class, your definition of `MailMessage` should include two methods that will only be used to make it simpler to define `getSummary` and `getHeadersAndText`. While `getSummary` and `getHeadersAndText` should be defined as public methods, the remaining methods should be defined as private.

**getHeaderContents**
This method will be an accessor method that returns a `String` value. It will take one `String` parameter which will be the word that appears at the start of the header line whose contents are desired. For example, to retrieve the contents of the Subject header of a `MailMessage` other methods in the class might say

```java
String subjHeader = this.getHeaderContents( "Subject" );
```

while to retrieve the From header you might use the code

```java
String fromHeader = this.getHeaderContents( "From" );
```

The method will determine what to return by applying the `indexOf` and `substring` methods to the instance variable that is associated with the collection of all header lines. The method will assume that this variable will have been set by the constructor of the class. The `String` returned should only include the text that appears after the colon and space that follow the word that identifies the type of the requested header. If the requested header line is not found, the method should return an empty string.

**getBoundary**
This method will be an accessor method that returns a `String` value. It will take no parameters. If any of the header lines of this message contain the substring “boundary=\n”, this method should return the `String` that appears between the quotes following `boundary=`. If no header line contains `boundary=`, then the method should return an empty string.

### Getting Started
You should begin your work this week by making a new copy of your project folder from last week, renaming it so that its name contains your OIT login id and “Lab5” (but no blanks). Open this project with BlueJ.

### Implementation Plan
As usual, you should plan to add code to implement one feature at a time and to test each feature before moving on to the next step. The following gives our suggested plan for such an approach.

**Almost implement the POPConnection class**
1. A good way to start this lab is by implementing a slightly simplified version of `POPConnection`. The simplification is that for now you should implement versions of the `getMessageHeaders` and `getMessage` methods that return `String` values rather than `MailMessage`es. This simplification is important for now because it makes it possible to define `POPConnection` before you define `MailMessage`. Later, it will be easy to revise the two methods to return `MailMessage` values.

Since this is the first time you will be writing a class that does not form a complete program by itself, we provide a “Working with Multiple Classes” appendix at the end of this handout to guide you through the process of using BlueJ to construct and test such classes. This appendix will lead you through the implementation of the methods of the `POPConnection` class.

**Modify TweetyBird to use the POPConnection class**
2. In last week’s lab, the `TweetyBird` class had an instance variable that was associated with a `NetConnection`. Your goal in this step is to replace that instance variable with a new variable that is a `POPConnection`. To accomplish this, you will need to replace each section of code in the `buttonClicked` and `menuItemSelected` methods with new (generally much shorter) code that depends on a `POPConnection` and its methods.

The changes in `buttonClicked` should be quite simple. The code we encouraged you to write for this method last week can be easily divided up into subsections that correspond closely to methods now provided by `POPConnection`. Unfortunately, converting `menuItemSelected` in this way is not as easy. This is because last week we encouraged
you to write a line-by-line loop to display the “interesting” headers of a method and then use a String loop to find the text/plain subsection. There is no way to process headers line-by-line using the methods of a POPConnection. With this in mind, we suggest you temporarily make your menuItemSelected method much simpler. In particular, we want you to revise menuItemSelected so that it works the way it did in Step 4 of last week’s implementation plan. This may seem scary, but recall that (if you followed our instructions above) you are working with a copy of last week’s lab. So, you will still be able to find the code you wrote in the final steps last week in last week’s project. With this in mind.

a. Replace the code in your menuItemSelected method so that when a menu item is selected your program displays all of the selected message’s headers and its entire body. That is, you just send the appropriate RETR command to the server and then use nextPOPResponse to get the complete contents of the message and place these contents in your JTextArea.

b. Ultimately, we want to replace the NetConnection instance variable you used last week with a POPConnection variable. If you do that immediately, however, none of the parts of your code from last week will even compile. So, we want you to do something a bit odd. Add an instance variable for the POPConnection but leave the NetConnection instance variable in place for now. This means you will have to choose a different name for the POPConnection variable.

This will make it possible to compile your program as you slowly make the changes in the next few steps. You should do this after each of the following steps and fix any syntax errors BlueJ reports. It will not, however, be possible to run the program successfully until all the steps in this subsection are complete. Just be patient.

c. Find the code in buttonClicked that makes a new NetConnection, sends the USER and PASS commands and then verifies that the PASS was accepted. Replace this code with instructions that create a new POPConnection and use its isConnected method to verify that the id and password were accepted.

d. Move on to the code that sends a STAT command through the NetConnection and extracts the number of messages available in the account. Replace this code with an invocation of getMessageCount on the POPConnection.

e. Find the loop that fills in your program’s menu. Within this loop there is code to send a TOP command through the NetConnection and retrieve the message headers using nextPOPResponse. Replace this code with an invocation of the getMessageHeaders method on the POPConnection.

f. Now, find the code in menuItemSelected that we had you simplify above so that it displays the entire message contents. Replace the lines that send the RETR command and uses nextPOPresponse with a line that invokes the getMessage method of the POPConnection.

g. At this point, you should not be using the NetConnection instance variable any more. Delete it. Now, run the program and, if necessary, debug the changes you have made.

h. Define and test a MailMessage class

3. As you did when writing the POPConnection class, you should try to first write and debug parts of the MailMessage class using the code pad before trying to incorporate the use of this new class into your POPConnection and TweetyBird classes.

a. Following the procedure described in the “Working with Multiple Classes” handout create a new class name MailMessage that does not extend GUIManager. Include two String instance variables that will shortly be associated with the headers and body of a mail message. Define a constructor that takes a String as a parameter and associates everything up to and including the first blank line in the parameter String with the instance variable you declared for the header and everything after the blank line with the instance variable for the body. Finally, define an initial, incomplete version of the getSummary method that just returns the entire header String as its result. Now, let’s test what you have written using the code pad.

b. To test the behavior of the MailMessage class in the code pad, you need a String that looks a bit like a sample message. With this in mind, the first things you should type in the code pad is a line to create a POPConnection followed by a line to associate the first message stored in the account used to create the POPConnection with a String variable name. These lines will look something like:
c. Now you should be able to create a new MailMessage by typing

```java
MailMessage msg = new MailMessage( sample );
```

and then test that the constructor has separated the headers from the full message by seeing whether typing

```java
msg.Summary()
```

returns the message's headers. This may be a bit hard to do because the result String will be displayed as one long line. Just scroll to the end of this String and make sure there is no message body.

d. Now define the getHeaderContents method. Eventually, this method should be defined as private. For now, however, make it public so that you can test it in the code pad.

e. When you think getHeaderContents is ready to test, repeat the commands shown above to set the variables toPOP, sample, and msg. Good news! You should not have to type in these lines again letter by letter. Using the up-arrow key should cause the code pad to scroll back through the earlier lines you typed. Once you find the right line, press return to re-execute it. After the three variables have been set, type "msg.getHeaderContents( "Date" )" to see if the correct String is returned.

f. Once it works for “Date”, verify that it works correctly for “Subject” and “From”. Then try it on a string like “Wierd” which will not appear as the first word in any header line and make sure it returns the empty string.

g. Once getHeaderContents seems to work, make it private and then revise getSummary to use it to return the contents of the From and Subject header lines. Test this new method.

**Modify TweetyBird and POPConnection to use the MailMessage class**

4. At this point, you can begin to incorporate the use of the MailMessage class in your POPConnection and TweetyBird classes.

   a. First, modify the getHeaders method of the POPConnection class so that instead of returning a String value, it returns a MailMessage created by applying the MailMessage constructor to the String it would have returned.

   b. The code in your TweetyBird class previously assumed that getMessageHeaders would return a String and it used indexOf and substring to form a message summary from this String. Now, you can modify this code to assume that getHeaders will return a MailMessage and use the getSummary method of the MailMessage class to obtain the summary that is added to the program's menu. Test your program once these modifications are complete.

   c. Define the getBoundary method of the MailMessage class. As you did for getHeaderContents, initially define it as public and use the code pad to verify its correctness.

   d. Once it works, make getBoundary private and then use it to define the getHeadersAndText method of the MailMessage class. Most of the rest of the code for this method can be produced by modifying the code you wrote last week for menuItemSelected.

   e. Modify the getMessage method of the POPConnection class to return a MailMessage.

   f. Finally, modify the menuItemSelected method in this week’s program to use the getMessage method of the POPConnection class and the getHeadersAndText method of the MailMessage class to determine what to display when a message is selected in the program’s menu.

**Clean Up**

Make sure to take a final look through your code, checking its correctness and style. Review the comments you received on the work you submitted in previous weeks and make sure you address the issues raised. Check over the style guide accessible through the course web page and make sure you have followed its guidelines. You should now be ready to submit your program.
APPENDIX: Working with Multiple Classes

Since this is the first time you will define several distinct classes as part of a single program, we will lead you through the initial steps in the definition of the POPConnection class to show how you can edit and test the separate classes that make up a multi-class program.

Add a minimal POPConnection class to your program

1. Click on BlueJ’s “New Class...” button. Provide the class name as usual, but do not select GUIManager as the type of class. Instead, use the setting “Class”. BlueJ will respond by creating a class definition with a skeletal constructor and method definition.

2. The “Class” template provided by BlueJ does not include any import directives. This particular class uses a NetConnection and the NetConnection class is part of the Squint library. So, you should insert

   import squint.*;

   as the first line of your POPConnection.java file.

3. Replace the comments in the first few lines of the new class with comments that describe the function of the POPConnection class. Do not include your name, even though a comment from the template tells you to. Instead, remove these comments from the template.

4. Next, replace the sample instance variable declaration for “x” with a declaration for a boolean variable named something like connected or loggedIn. This variable will have roughly the same role as the similar variable we told you to define in TweetyBird last week. Update the comment that describes the variable while you are at it. Replace the instruction in the constructor to set x equal to 0 with an assignment to make your new instance variable equal to true.

5. Now, we are going to have you define a silly, preliminary version of the logout method. This method is not expected to return any result, so it should be defined as

   public void logout() { ... }

   Add such a definition after the code for the “sampleMethod” included in the template. (Don’t delete sampleMethod. We will convert it into a method you want in just the next step.) Eventually, the body of the logout method should send information to the server to end the connection. For now, just include an assignment statement that sets the connected/loggedIn variable to false.

6. Next, define the isConnected method described earlier in the handout. This method should simply return the current value of your connected/loggedIn variable. You can use the sampleMethod included in the template to save a little time defining isConnected. First, change the method name to isConnected and change the return type from int to boolean. Delete the parameter declaration for y since isConnected does not expect any parameters. Then, replace the “x+y” in the return statement with the name of your boolean instance variable. DONE!

7. Press “Compile” and fix any errors you might have made following our instructions so far.

Creating an instance of your POPConnection class in the Code Pad

Let’s test that the code you have entered works correctly so that you can learn how to test a class like this before adding any more complex code to the program.

8. Click on the BlueJ project window. An image of what it should look like is shown at the top of the next page. If you don’t see the area labeled as the code pad in our screen capture, select the “Show Code Pad” item from the BlueJ “View” menu. A new area known as the code pad will appear.

9. Resize the project window and its subparts as necessary to make the code pad larger so that things look a bit like the image at the top of the next page.

10. Within the code pad, type an assignment statement to associate a local variable name with a new POPConnection. This will be easy because at this point the constructor in your minimal version of the class expects no parameters. Therefore, you should be able to just type something like
CS 134

11. Now, to see if your simple methods are working correctly, type

```java
toServer.isConnected()
```

in the code pad. The value produced by this line should be true, because your constructor assigned this value to the connected/loggedIn instance variable.

NOTE: The line we had you type to create the POPConnection ended with a semicolon but the line to test isConnected has no final semicolon. When working in the code pad, if you type some Java code with the hope of seeing the value/answer the code produces, there should be no semicolon at the end of the line. On the other hand, if you type a line to assign a value to a variable or change the state of some object, there should be a semicolon at the end of the line.

12. Now, enter the two lines

```java

toServer.logout();
toServer.isConnected()
```

The result of the second line should be false this time, because the logout method is defined to change the information associated with the POPConnection’s instance variable.

**Add code to make your POPConnection really log in and log out**

In your final version of POPConnection you will need to do much more than set a boolean variable true or false. In particular, the constructor will need to create a NetConnection to an actual POP server and send USER and PASS commands through this connection. The logout method will have to send a QUIT command through the same connection and then close it. With this in mind.

13. Add a second instance variable of type NetConnection to your class.

14. Add three String parameter declarations to the POPConnection constructor that will provide the name of the server to which it should connect along with the user id and password of the account to use.

15. Add code to the POPConnection constructor to make a new NetConnection and send appropriate USER and PASS commands. At the end of this code, the constructor should set the connected/logged variable to true if the response to the PASS command stated with +OK and it should set this variable to false otherwise.
16. Add code to the logout method to send a QUIT command to the server and to close the NetConnection. Keep the line that sets the connected/loggedIn variable to false.
17. Compile your new code and test that the constructor, isConnected method and logout methods all work as expected using the code pad.

**Complete the POPConnection class**
18. Now, continue in the same manner to define and test the getMessageCount, getMessage, and getMessageHeaders methods as described in the lab handout.