Lab 9

──── Spam, Spam, Spam ────

**Objective**   To gain experience with Strings.

You may work with a partner if you wish.

Before the mid-90s, Spam was a canned "meat" product. These days, the term "spam" means something else — unwanted email.

This week we build a program that provides insights into how mail programs filter spam. Our spam filter is rather simple. The user provides a list of words called the *filter list*. The program searches the "from" and "subject" headers of all your mail messages and divides your mail into a *good list* and a *spam list*. The spam list contains all the messages that contain one or more words from the filter list, and the good list contains the messages that do not contain any filter words.

The user interface for the SpamFilter program is shown below:
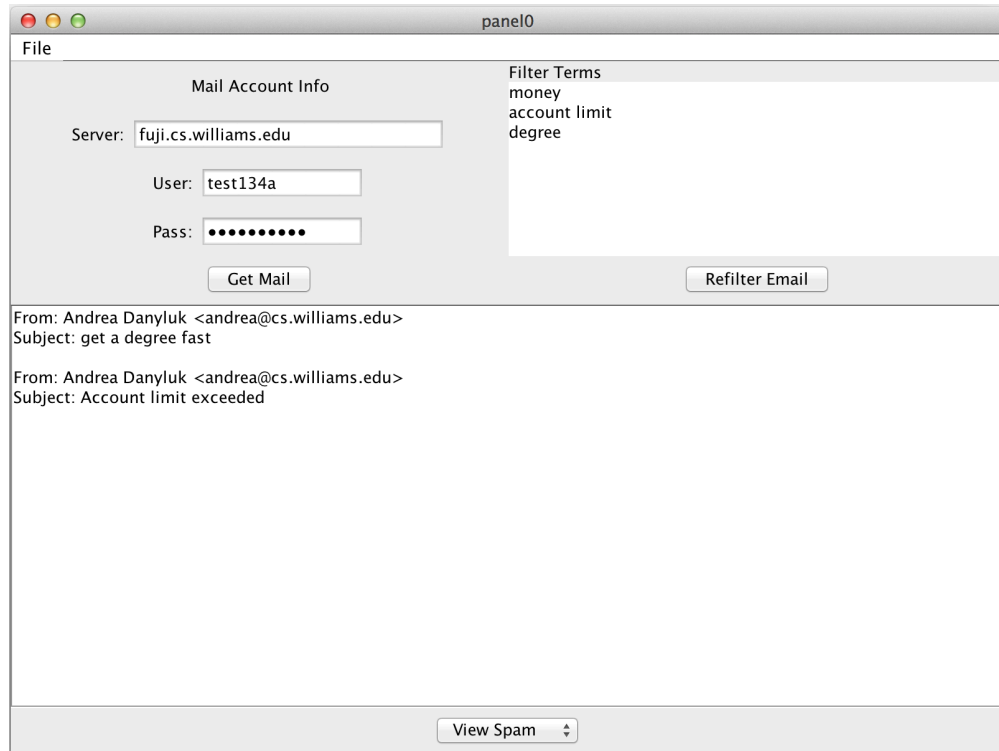
The text fields on the top left side of the window specify the machine and account from which mail will be fetched. When the "Get Mail" button is pressed, the program connects to the mail server and downloads the headers of all available messages.

The program then decides whether each fetched message is a good message or spam, based on what the user enters in the `JTextArea` labeled "Filter Terms." (`JTextArea`s are just like `JTextField`s except that they display more than one line of text.) Each line of text entered in this `JTextArea` is treated as an indicator of spam. If the "from" field or the "subject" field of a mail message contains a substring that is identical to one of these lines, the message is considered spam.

The `JComboBox` at the bottom of the screen allows the user to switch between displaying the good mail or the spam in the `JTextArea` that occupies the center of the program window.

You can use this program to connect to mailboxes we have set up for you on the mail server `fuji.cs.williams.edu`. You will connect to that server with the same username and password that you use to log in on our lab machines. These mailboxes are currently empty, and you will need to send yourself a few short pieces of mail to test out your program. Your address will have the form `abc3@fuji.cs.williams.edu`, where `abc3` is replaced with your user name.

We provide the code to actually communicate with a mail server, and our code does not modify the mailbox on the server in any way.

If using your own computer, PurpleAir may prevent you from connecting to our server. You will need to work on a lab computer if you're unable to connect to the server.

Program Structure

Your program will consist of 4 main classes. We will provide one of those (`MailConnection`) as a library, so you won't see it in the BlueJ project window.

SpamFilter. This will be the controller class for your program, and we will provide all of the code for it. This class creates the components of the interface and defines event-handling methods for them. Since this program does not use the canvas, it extends `Controller` instead of `WindowController`.

MailConnection. This class provides methods to contact a mail server through the network and download messages from the server. As noted above, we will provide the code for this class as a library.

The `MailConnection` class provides several methods that are used by the `Controller`.

- `public MailConnection(String host, String userName, String password)`: The constructor expects three `String`s as parameters. The first must be the name of the mail server to contact. When using the mailbox we created for you, the `host` will be `fuji.cs.williams.edu`, and the `userName` and `password` parameters will be what you use to log in to our lab computers.

- `public boolean isConnected()` Returns true if the program currently has a connection to the mail server.

- `public void disconnect()` Closes the connection to the mail server. This does nothing if there is no active connection.

- `public int getNumMsgs()` Returns the number of messages in the mailbox you are connected to. This returns 0 if there is no active connection.

- `public String header(int msg)` Returns the headers of a mail message identified by the number passed in. Unlike Java, mailboxes number messages beginning with 1 and going up to the number of messages contained in the mailbox.

  The mail headers are returned in one long string, such as:

  ```
  Return-Path: <lvining@cs.williams.edu>
  Received: from [137.165.8.62] (tcl303.cs.williams.edu [137.165.8.62])
  by bull.cs.williams.edu (8.12.3p3/8.12.3) with ESMTP id hADDU9NG013481
  for <colloquium@cs.williams.edu>; Thu, 10 Nov 2016 08:30:09 -0500 (EST)
  (envelope-from lvining@bull.cs.williams.edu)
  Mime-Version: 1.0
  Message-Id: <a05200f02bbd9370948a5@[137.165.8.62]>
  Date: Thu, 10 Nov 2016 08:30:01 -0500
  To: colloquium@cs.williams.edu
  From: Lauren Vining <lvining@cs.williams.edu>
  Subject: colloquium
  Content-Type: multipart/alternative; boundary="=======_-1143392286==_ma======="
  Status: RO
  ```

  Your spam filter will look at only the "From" and "Subject" headers. Part of your job, described below, is to extract just those headers from the long header list that `header` returns.

  The `header` method returns an empty string if it is called when there is no connection.

Message. You will construct a `Message` object for each mail message downloaded from the server. The class is simple. It has a constructor and three accessor methods:

- `public Message(String headers)`: The constructor expects a `String` containing the message (or at least its header) as a parameter

- `public String getFrom()`: Returns the "From:" line found in the header of this message.

- `public String getSubject()`: Returns the "Subject:" line found in the header of this message.

- `public String toString()`: Returns the "From" and "Subject" headers as a single string with a newline between them and a newline at the end.

3

MessageList. A `MessageList` holds a list of mail `Message`s. Internally, it uses an array to keep track of the members of the collection. To make it possible to create an array of an appropriate size, the constructor takes the maximum size of the collection as a parameter.

We provide the necessary constructor and an `add` method to add messages to a message list. You should write three methods:

`public String toString()`: This method invokes the `toString` method of the `Message` class to obtain `String`s describing each of the messages in the list. It concatenates all of these descriptions together (separated from one another by extra newlines) and returns this large `String` as its result.

`public MessageList getSpam(String[] filterWords)`: This method takes an array of `String`s containing one `String` for each line entered in the filter `JTextArea`. It returns the spam. To do this, it creates a new `MessageList` and adds to it any messages that are spam.

`public MessageList getGood(String[] filterWords)`: This method should take an array of `String`s containing one `String` for each line entered in the filter `JTextArea`. It does the same thing as `getSpam`, except that it returns the good messages.

Suggestions For Designing These Classes.

- Extracting the "From" and "Subject" headers from the long string that `header` returns is part of your task. As shown earlier, the String that `header` returns actually contains multiple headers with a newline between each. To find just one header, you should find a string that begins with a newline character `\n` followed by "From:" or "Subject:" and ending at the next newline character. Be sure to handle the special case where the header you are looking for is the last header and does not end with a newline! You should use a case-insensitive comparison when looking for these strings, but the string returned from this method should be capitalized as in the original header.

- You will need to determine if a header is spam. You should use `String` methods to search the header for the presence of any `String` in the filter list. You should use a case-insensitive comparison for your spam comparisons.

  Note that the user may have inadvertently added blank lines to the filter area. When you look for matches in the filtering methods, you should ignore any empty strings in the array of filter terms.

- You may find it useful to introduce other private methods to keep your code simple and to prevent repetitive code in several places.

Design

This week we will again require that you prepare a written design for your program before lab. At the beginning of the lab, the instructor will briefly examine each of your designs to make sure you are on the right track.

Your design should include the following:

1. A design for the `Message` class and its three methods.

2. A design for the three methods in the `MessageList` class. We suggest that you think about writing an additional method in `MessageList` that you can use in both filtering methods with the following declaration:

   `private boolean matchExists(String[] terms, String searchString)`

   This method will return true if any of the terms appear in `searchString`.

The more time you spend on your design, the faster you will be able to proceed.

Implementation Strategy

We suggest that you approach this problem in the following order:

- Since your temporary mailbox on fuji is currently empty, send yourself a few short pieces of email.

- Download the starter code for this project from the handouts web page.

- Define the `Message` class. The `SpamFilter` class is initially set up to use a `MailConnection` to download the first message's header. This header string is used to create a new `Message`, and the `SpamFilter` then displays the results of calling `getFrom`, `getSubject`, and `toString` on that message. Once you implement the `Message` methods, the appropriate information should be displayed for it in the text area. However, none of the filter components will behave properly yet.

- Once you finish the `Message` class, open the `SpamFilter` class and add the following code where it says "INSERT CODE HERE":

```
int numMessages = connection.getNumMsgs();
allMessages = new MessageList(numMessages);
for (int i = 0; i < numMessages; i++) {
    Message m = new Message(connection.header(i + 1));
    allMessages.add(m);
}
filterMessageList();
```

This will make the spam filter download all of the messages and then filter the message list according to which filter is currently selected. You will also need to comment out the lines before it that downloaded the first message for the previous step.

- Start working on the `MessageList` and write the `toString` method. Once the `toString` method is implemented, the from and subject lines for all messages downloaded from the server should appear in the window when you get your mail.

- The last step is to write the two filtering methods. The starter code in the message list contains a `getGood` that returns a message list of all messages and `getSpam` that returns an empty message list. Change the filter methods in the message list to work properly.

We also include some ideas for optional extensions at the end of this handout.

Submitting Your Work

Once you have saved your work in BlueJ, please perform the following steps to submit your assignment:

- First, return to the Finder. You can do this by clicking on the smiling Macintosh icon in your dock.

- From the "Go" menu at the top of the screen, select "Connect to Server...".

- For the server address, type in "afp://Guest@fuji" and click "Connect".

- A selection box should appear. Select "Courses" and click "Ok".

- You should now see a Finder window with a "cs134" folder. Open this folder.

- You should now see the drop-off folders for the three lab sections. Drag your "Lab9Spam" folder into the appropriate lab section folder. 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. on Wednesday if you're in the Monday night lab; up to 6 p.m. on Thursday if you're in the Tuesday morning lab; and up to 11 p.m. on Thursday if you're in the Tuesday afternoon lab. If you submit and later discover that your submission was flawed, you can submit again. We will grade the latest submission made before your lab 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 "revised") and the re-submission will work fine.

Grading Guidelines

As always, we will evaluate your program for both style and correctness. Here are some specific items to keep in mind and focus on while writing your program:

Style
Descriptive comments
Good names
Good use of constants
Appropriate formatting

Design
General correctness/design/efficiency issues
Conditionals and loops
Parameters, variables, and scoping
Correct use of arrays
Good use of private methods
Good use of strings

Correctness
Extracting from and subject headers
Displaying message list properly
getGood filter
getSpam filter

You will write the `Message` class from scratch, and the following starter code will be given to you for `MessageList`. You may examine the `SpamFilter` code by downloading the starter project, but you should not need to read that file to complete your design.

```java
public class MessageList {

    // array of message objects to keep the list
    private Message messages[];

    // number of entries being used in the messages array
    private int count = 0;

    // Constructs a new message collection with the given maximum size.
    public MessageList(int maxSize) {
        messages = new Message[maxSize];
    }

    // Add a new message to the list.
    public void add(Message newMessage) {
        if (count < messages.length) {
            messages[count] = newMessage;
            count++;
        } else {
            System.out.println("No More Space in Message List!!!");
        }
    }

    public String toString() {
        return "Complete MessageList.toString()";
    }

    /*
     * CHANGE THIS METHOD
     * It currently just returns all of the messages.
     */
    public MessageList getGood(String[] filterTerms) {
        MessageList resultList = new MessageList(messages.length);
        for (int i = 0; i < count; i++) {
            resultList.add(messages[i]);
        }
        return resultList;
    }

    /*
     * CHANGE THIS METHOD
     * It currently just returns an empty list.
     */
    public MessageList getSpam(String[] filterTerms) {
        return new MessageList(0);
    }
}
```

▬▬▬ Optional Extensions ▬▬▬▬▬▬

If you wrap up early and would like to extend the SpamFilter, here are two ideas.

Thresholds. Most spam filters only filter mail whose "spam content" is deemed high. A message with a single spam word may not be spam, but one with perhaps 3 or more almost certainly is. This helps reduce false positives, in which good mail is marked as spam.

Add a threshold to your filter. In particular, change your program so that it only marks a message as spam if its headers contain n or more occurrences of spam words. You should change the SpamFilter's interface so that the user can configure the threshold n to anywhere between 1 and perhaps 5. (A JSlider might work well for this.)

Experimenting with Inheritance. Your program's interface includes several JTextFields each of which must be placed next to a JLabel describing its function. Rather than creating each of these separately in your program's begin method, define a new class of GUI components that combines a JLabel and a text field in such a way that you can treat them as a single GUI component.

You can create a new class of GUI components that can hold several other components together by defining a class that extends JPanel. To understand how this works, think about what you would have done if you decided to create the needed JLabels and JTextFields separately in your begin method. To ensure that they were kept together in the program's interface you would create a JPanel and add them both to the JPanel (which would then in turn be added to the content pane).

Suppose now, that you instead want to define a new class named LabeledTextField designed to combine a JLabel and a JTextField. If you define the new class as an extension of JPanel this is quite easy. Since any object of the new class is a JPanel, Swing will let you add the object to the content pane. Also, since the object is a JPanel, you can include code in the constructor for the new class that creates a JLabel and a JTextField and adds them to the object being created. Finally, if you associate the JTextField with an instance variable then you can define a getText method for the new class which simply returns the result of invoking getText on the JTextField associated with the instance variable.

Alas, things are made a bit more complicated by the fact that your program requires two JTextFields and one JPasswordField, all of which need to have JLabels attached to them. You could just define two separate classes that extend JPanel as described above, but to maximize your exploration of inheritance we have something more elegent in mind.

First, define a class named LabeledTextComponent. This class has a single purpose: to be extended by LabeledTextField and LabeledPasswordField. The LabeledTextComponent class should extend JPanel. It should include an instance variable of type JTextField and a getText method that returns the String obtained by invoking getText on the instance variable. Its constructor, however, will not actually create a text field. Instead, all it will do is create a JLabel and add it to the JPanel. As a result, the only parameter to the constructor for this class will be the String to use when creating the JLabel. Thus, the following is a complete implementation of LabeledTextComponent:

```
class LabeledTextComponent extends JPanel {

    protected JTextField textField;

    public LabeledTextComponent(String label) {
        add(new JLabel(label));
    }

    public String getText() {
        return textField.getText();
    }
}
```

You will define the two desired classes, `LabeledTextField` and `LabeledPasswordField` as extensions of `LabeledTextComponent`. Each of the constructors for these two classes will take an integer that determines how wide the field should be and a `String` that determines what is initally displayed in the field, as well as the label necessary for the super class. The constructor for `LabeledTextField` will create a `JTextField` and add it to the object being constructed. The constructor for `LabeledPasswordField` will create a `JPasswordField` and add it to the object being constructed. Both classes will inherit `getText` from `LabeledTextComponent`.

Add these three classes, and change the `SpamFilter` to use them when building the interface components in `begin`.