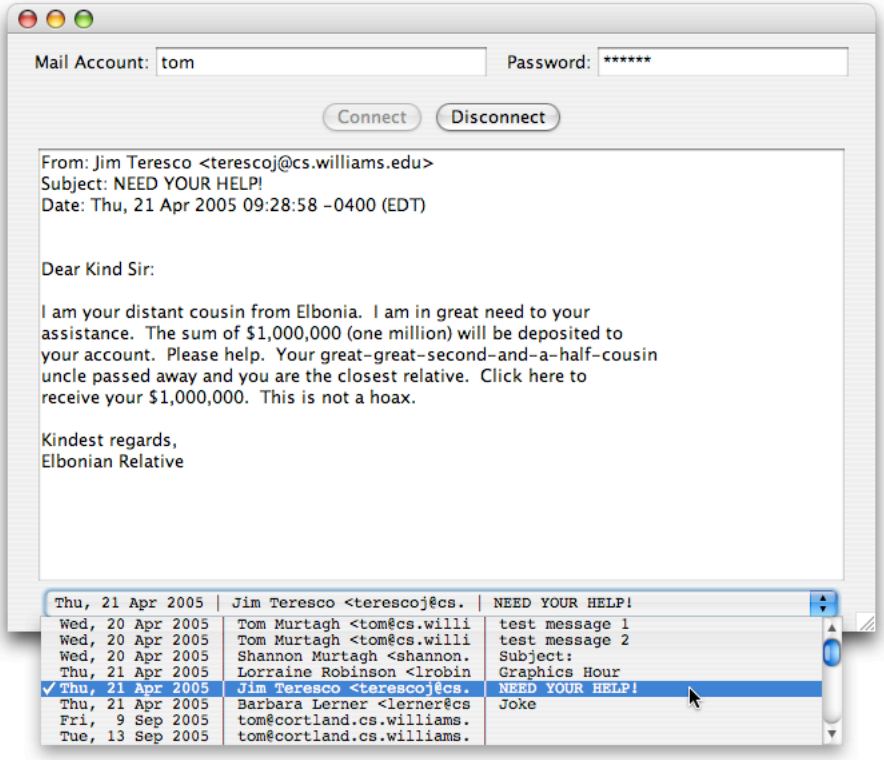
Lab 6

Recursive Revision

List Based Mail Client
Due March 13/14 11PM

This week, we want you to write yet another POP client. Externally, this program will look quite a lot like the program you completed for Lab 3. A sample of its interface is shown below. The components at the top of the program window are identical to those used in Lab 3. There are JTextFields and JButtons used to log in and out of a POP server and a JTextArea used to display a requested message. The mechanisms provided to request that a particular message be displayed, however, are quite different.



The program you complete this week will provide a menu that can be used to determine which message will be displayed. The items in this menu will be summaries of the email messages available in the user's account. Each summary will include the date the message was received, the name of the person who sent the message and the subject field (if any) included with the message. When the user selects an item in this menu, the program should display the corresponding email message in its JTextArea.

A Helping Hand

In past labs, you have either written an entire program from scratch or extended a program you had started the previous week. You certainly might imagine completing this week's assignment by starting with the program you completed for Lab 3. In fact, we are going to make things even easier. As a starting point this week, we will provide you with a complete Java program that you can download from the course website that functions exactly like the program we want you to submit!

What's the catch?

Well, while we don't want you to change the way the program we give you behaves externally, we want you to change how it is implemented internally. In particular, we want you to change two of the classes so that they use recursive structures.

While there is a catch, it is the case that the code we are providing should make it easier for you to complete this lab. We recognize that this will be a busy week. Among other things, our course midterm is this week. With this in mind, we have tried to design this lab so that you can focus your attention on the new programming topic we have just covered in class, recursion, while wasting as little effort as possible on other programming tasks. The code we provide should spare you from having to worry about user interfaces issues, etc. All you have to do is implement two recursive classes and change a few lines of our code to use them.

Of course, there is another catch. Since we have reduced what we ask you to do for this lab, we will give you less time to do it. The deadlines for this week's lab are earlier than usual. Students in the Monday lab should complete the program by Tuesday at 9PM and students in the Tuesday lab should finish by Wednesday at 9PM. Guess when we will be holding our midterm review session!

A Quick Tour

The program we will provide you is divided into three classes named `POPClient`, `MailMessage`, and `POPConnection`. A complete listing of the code for these classes is attached to this handout. We will provide an overview of the structure of each of these classes here.

POPClient

The POPClient class displays the program's user interface and reacts to user requests. Its constructor creates the text fields, text area, and buttons included in the interface and associates them with instance variables. It includes the definition of three event-handling methods, `buttonClicked`, `textEntered` and `menuItemSelected`.

The `buttonClicked` method handles the process of logging in and out from the POP server. The code that performs the login process differs from the corresponding code in your program from Lab 3 in two important ways. Rather than explicitly send the "USER" and "PASS" commands to the server, the code we have provided depends on a method named `login` included in a separate class named `POPConnection` to perform these steps. In addition, after the login is complete, the code in our `buttonClicked` method immediately retrieves all of the messages available on the account, extracts summaries of these messages, and builds a menu containing one summary line for each message. This menu is then added to the program's window.

The code to handle the "Disconnect" button is somewhat simpler. It simply logs out from the server and removes the message menu from the display. Like the login code, it does not explicitly send any messages (the "QUIT" command) to the server. Instead, it depends on a method provided by the `POPConnection` class.

The `menuItemSelected` method contains code to fetch a message from the server when the user selects a new message summary from the menu created at login. This code also depends on a method of the `POPConnection` class to retrieve messages. Therefore, it does not explicitly send "RETR" commands to the server.

The `textEntered` method provides a shortcut to pressing the "Connect" button.

POPConnection

The `POPConnection` class provides four methods that can be used to perform basic interactions with a POP server. The class constructor expects no parameters. You can construct a new `POPConnection` with

```
POPConnection toServer = new POPConnection();
```

Constructing a `POPConnection`, however, does not actually cause your computer to connect to a server. To do this, you must use the `login` method. This method expects three parameters: the server to contact, the account username and the account password. It returns a boolean value indicating whether the login was successful. The method is typically used as follows:

```
if (toServer.login(SERVER_NAME, userId, password)) {

     // code to complete connection

} else {

    // code to inform user of login failure

}
```

Once logging in successfully, you can use two methods to access information about the account. The method `messagesAvailable` returns the number of messages currently stored on the account as an `int`. The method `getMessage` takes a message number as a parameter and returns the requested message. The message number parameter must be an `int`, and the message returned is an object of the `MailMessage` class described below.

Finally, the close method may be invoked to log out from a POP server.

MailMessage

The `MailMessage` class is designed to provide convenient access to the components of a message retrieved from a POP server. Its constructor takes no parameters and creates an "empty" mail message. Like the Swing `JTextArea` class, it provides an `append` method that can be use to add lines to the `MailMessage` as they are received from the server.

Unlike the `MailMessage` class described in lecture, the `MailMessage` class here stores the message as two separate Strings associated with distinct instance variable names. The name headers is associated with a String containing all of the header lines that precede the message body and the name contents is associated with the message body itself.

Once all of the lines of a message have been added using the append method, two messages can be used to access the contents of the message. The `toString` method will return then entire contents of the body of the method preceded by its most important header lines (To, From, Date, and Subject). The `shortSummary` method returns a single line containing parts of the Date, From, and Subject fields of the message suitable for use as an item in the message menu.

To simplify the definition of the `toString` and `shortSummary` methods, the class includes three private methods named `getHeader`, `truncatedHeader`, and `shortHeaders`. The `getHeader` method takes the name of a header line ("Date: ", "From: ", etc.) and returns the corresponding header line for the message. The `truncatedHeader` method is like the `getHeader` method but it takes a second parameter that determines the length of the string returned. The header will either be truncated or padded with blanks so that it has the desired length. The `shortHeaders` method returns a String containing the From, To, Date, and Subject header lines.

Your Task

We want you to make two changes to the classes we have provided.

StringList

The `MailMessage` class keeps track of the lines of a message using two String variables named headers and contents. Each of these Strings typically holds multiple lines separated by "\n" characters. We would like you to define a recursive class named `StringList` that can be used to represent such a collection of lines. Each object in a recursive String list should include a String corresponding to a single line of text and another `StringList` representing the remaining lines in the collection. Once you have defined the `StringList` class, we want you to change the declarations of the two variables named headers and contents to be `StringLists` rather than Strings. Then, you should make whatever other changes are necessary in the definition of the `MailMessage` class that are required given this new way of representing the headers and message body. You should discover that relatively few changes will be required to do this. In particular, you should only have to modify the `MailMesage` constructor, the `append` method and the `getHeader` method.

To make this possible, your `StringList` class should include two constructors and two methods. One constructor should take no parameters and return an empty `StringList`. The other constructor should take a String and a `StringList` and construct a new, bigger `StringLis`t that includes the new line in addition to all the lines in its `StringList` parameter.

The first method you should define is a `toString` method. This method should return a String formed by concatenating all of the lines in the `StringList` together separated by "\n" characters. You will need to be a bit careful when writing this method to make sure that the lines appear in the correct order. The second method should be named `getLineStartingWith`. It should take a `String` as a parameter and return a line from the `StringList` that starts with the parameter `String`. This method should return the empty string "" if no match is found.

MessageList

The second recursive class we want you to define will be used to hold a collection of `MailMessages`. Each of the `MailMessages` in the collection will be paired with the message number used to fetch that message from the POP server.

This class will also provide two constructors. One constructor will take no parameters and return an empty `MessageList`. The other constructor will expect three parameters: a `MailMessage`, and an `int` representing that message's sequence number, and an existing `MessageList`. It will form a larger `MessageList` by adding the `MailMessage` and its sequence number to the collection.

The `MessageList` class will provide just a single method named `get`. This method will behave much like the `get` method provided by the `HashMap` class. It will take a message number as a parameter and, if possible, return the message associated with that sequence number. If no matching message can be found in the `MessageList`, it should return null.

Once this class is written, you should use it to modify our `POPConnection` class in an interesting way. The goal will be to use a `MessageList` to implement a "cache" of messages that have already been fetched from the server. Every time `POPConnection` fetches a message from the server, it will add this message and its message number to this `MessageList`. Then, whenever it is asked to get a message, the `POPConnection` will first use the `get` method to see if the a message with the desired message number is already in the `MessageList`. If so, it will simply return the message provided by `get`. Otherwise it will fetch the message and add it to the `MessageList`.

Start by declaring a `MessageList` as an instance variable in the `POPConnection` class and associating this variable with an empty `MessageList` as part of the `login` method. Then, rename our `getMessage` method to `fetchMessage`, change it from public to private, and add code to add each message fetched to the `MessageList` cache. Finally, implement a new, public `getMessage` method. This method will first

use get to see if the desired message is already in the cache. If so, it will simply return it. If not, it will use `fetchMessage` to access the message and return the result.

In the context of the `POPClient` we have provided, this change will make the program much more efficient. Since our client first fetches all available message to build the message menu, all the message will end up in your `MessageList` cache. As a result, when the user actually selects a message from the menu, it will be displayed without any additional network traffic.

Implementation Plan

- Begin by reading the existing code in RecursiveRevision lab. It is attached at the end of this handout, but you should also download it from the home page: http://www.cs.williams.edu/~cs134/s07/labs.html.
- Once you're familiar with the existing code, create a new class called `StringList`. Make sure to create two constructors: one with no arguments that creates an empty `StringList`; and a second with two arguments that creates a `StringList` from a `String` and an existing `StringList`.
- Now add the `lineStartingWith` method and the `toString` method. Test both constructors of your `StringList` by creating two instances of your class: one for the empty constructor and one for the constructor that takes a `String` and a `StringList`. For the second argument of the StringList constructor, you can use an empty string list by typing "`new StringList()`". Use this second constructor to test the `lineStartingWith` and `toString` methods.
- Modify the `MailMesage` constructor, the `append` method and the `getHeader` method so that the `MailMessage` class behaves as before. Remember that the contents and header member variable now have type `StringList` instead of `String`.
- Now create a new class called `MessageList`. It should have two constructors: one takes no parameters and returns an empty `MessageList`. The other constructor takes three parameters: a `MailMessage`, an `int` representing that message's sequence number, and an existing `MessageList`.
- Add a `MessageList` member variable called `cache` in the `POPConnection` class. Make sure to associate `cache` with an empty `MessageList` as part of the `login` method.
- Rename the `getMessage` method to `fetchMessage`, change it from public to private, and add code to add each message fetched to the `MessageList` cache.
- Implement a new, public `getMessage` method. This method will use `get` to see if the desired message is already in the cache. If it is, return it. If not, use `fetchMessage` to access the message and return the result. Make sure that if `fetchMessage` is successful that you update the `cache`. To do this you'll need to set `cache` to be a new `MessageList` composed of the recently fetched `MailMessage` and the previous `MessageList`.
- At this point, the internal structure of your program should be radically different from the start, but the behavior of the program should be exactly the same.

Clean Up

Make sure to take a final look through your code checking its correctness and style. Check over the style guide accessible through the course web page and make sure you have followed its guidelines. Make sure you included your name and lab section in a comment in each class definition.

Grading

Completeness (14 pts) / Correctness (6 pts)

- Correct constructors for StringList
- toString includes lines in correct order
- getLineStartingWith implemented
- Message class modifed appropriately
- MessageList constructors
- Correct constructors for MessageList
- MessageList get method
- POPClient modifed appropriately

Style (10 pts)

- Commenting
- Good variable names
- Good, consistent indentation
- Good use of blank lines
- Removing unused methods
- Uses public and private methods appropriately

Submission Instructions

Find the folder that BlueJ created for your project. Its name should be the one you picked for your project (something like FloydLab6).

- Click on the Desktop, then go to the "Go" menu and "Connect to Server."
- Type "cortland" in for the Server Address and click "Connect."
- Select Guest, then click "Connect."
- Select the volume "Courses" to mount and then click "OK." (and then click "OK" again)
- A Finder window will appear where you should double-click on "cs134",
- Drag your project's folder into either "Dropoff-Monday" or "Dropoff-Tuesday".
- Log off of the computer before you leave.

You can submit your work up to 9 p.m. one day after your lab (9 p.m. Tuesday for those in the Monday Lab, and 9 p.m. Wednesday for those in the Tuesday Lab). If you submit and later discover that your submission was flawed, you can submit again. The Mac will not let you submit again unless you change the name of your folder slightly. Just add something to the folder name (like the word "revised") and the re-submission will work fine.

```
import squint.*;
import javax.swing.*;
import java.awt.Font;

/**
 * POPClient --- This program allows its user to view
 * mail messages accessed through a POP server.
 */
public class POPClient extends GUIManager {

    // Change these values to adjust the size of the program's window
    private final int WINDOW_WIDTH = 650, WINDOW_HEIGHT = 470;

    // The server to use
    private final String SERVER = "cortland.cs.williams.edu";

    // The standard POP port
    private final int POP_PORT = 110;

    // User interface buttons
    private JButton login = new JButton( "Connect" );
    private JButton logout = new JButton( "Disconnect" );
    private JButton request = new JButton( "Get message" );

    // Used to enter the POP account identifier
    private JTextField user = new JTextField( 20 );

    // Used to enter the POP account password
    private JPasswordField pass = new JPasswordField( 15 );

    // Email messages are displayed in this area
    private JTextArea message = new JTextArea( 20, 50 );

    // +OK and -ERR messages from the server are displayed in this area
    private JTextArea log = new JTextArea( 5, 50 );

    // Our connection to the POP server
    private POPConnection toServer;

    // Menu used to select messages;
    private JComboBox messageSelector;

    private boolean connected = false;

   /*
    * Install all of the required GUI components
    */
    public POPClient() {
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

        // Each JLabel/JTextArea pair is placed together in a panel of their own
        JPanel curPanel;

        // Initial button states
        login.setEnabled( true );
        logout.setEnabled( false );
        request.setEnabled( false );

        // Create fields for entering the account information
        curPanel = new JPanel();
        curPanel.add( new JLabel( "Mail Account:" ) );
        curPanel.add( user );
        contentPane.add( curPanel );

        curPanel = new JPanel();
        curPanel.add( new JLabel( "Password:" ) );
```

```
        curPanel.add( pass );
        contentPane.add( curPanel );

                // Create a field for the message number
        curPanel = new JPanel();
        curPanel.add( login );
        curPanel.add( logout );
        contentPane.add( curPanel );

        // Install the retrieve button and text areas in the window
        contentPane.add( new JScrollPane( message ) );
        //contentPane.add( new JScrollPane( log ) );
    }

   /*
    * When the button is clicked, interact with the POP server to
    * access the requested message.
    */
    public void buttonClicked( JButton which ) {
        String serverResponse;

        if ( which == login ) {
            // Connect to the server and display initial response
            toServer = new POPConnection();
            if (toServer.login(SERVER, user.getText(), new String(pass.getPassword()))) {
                int totalMessages = toServer.messagesAvailable();

                messageSelector = new JComboBox();
                messageSelector.setFont( new Font( "Courier", Font.PLAIN, 12 ) );
                int messNum = 1;
                while ( messNum <= totalMessages ) {
                   messageSelector.addItem(toServer.getMessage(messNum).shortSummary() );
                   messNum = messNum + 1;
                }
                contentPane.add( messageSelector );
                connected = true;
            } else {
                message.setText( "Unable to login.  Check you password." );
            }

        } else if ( which == logout ) {
            // Terminate the connection
            toServer.close();
            contentPane.remove( messageSelector );
            connected = false;
        }

        login.setEnabled( ! connected );
        logout.setEnabled( connected );
        request.setEnabled( connected );
        repaint();
    }

    /**
     *  Request the selected message from the server
     *  and display it in the text area.
     **/
    public void menuItemSelected() {
       MailMessage requested;
       requested = toServer.getMessage(messageSelector.getSelectedIndex()+1 );
        if ( requested == null ) {
            message.setText( "Unable to retrieve message" );
        } else {
            message.setText( requested.toString() );
            message.setCaretPosition(0);
        }
```

```
     }

     /**
      * Simulate clicking the login button
      */
     public void textEntered() {
         login.doClick();
        }

}


---------------------

/**
 * This class represents an email message.  It provides methods
 * to retrieve the full header, the short header (To, From, Date, Subject),
 * a truncated header, and several 'pretty' views of the mail message.
 */
public class MailMessage {

    private String contents;
    private String headers;

    /**
     * Construct an empty mail message
     */
    public MailMessage( ) {
        headers = "";
        contents = null;
    }

    /**
     * Add new lines to the mail message.  The first lines added
     * are for the header.  The later lines denote content.  The header
     * and content are separated by an empty string.
     */
    public void append( String newLine ) {
        if ( contents == null ) {
            if ( newLine.equals("") ) {
                contents = newLine;
            } else {
                headers = headers + newLine + "\n";
            }
        } else {
            contents = contents + newLine + "\n";
        }
    }

    /**
     * Return the 'prefix' header of the message.
     * For example, 'prefix' might be the "To:" or "From:"
     * field of the header.
     */
    private String getHeader( String prefix ) {
        int start = headers.indexOf( prefix );
        if ( start >= 0 ) {
            int end = headers.indexOf( "\n", start );
            return headers.substring( start, end );
        } else {
            return "";
        }
    }

    /**
     * Return a truncated header for 'prefix' with length
```

```
     * at most 'len'.
     */
    private String truncatedHeader( String prefix, int len ) {
        String result = getHeader( prefix );
        if ( result.length() > prefix.length() ) {
            result = result.substring( prefix.length() );
        }
        while ( result.length() < len ) {
            result = result + "     ";
        }
        return result.substring( 0, len );
    }

    /**
     * Return the standard header fields, separated by newlines.
     */
    private String shortHeaders() {
        return getHeader( "From: " ) + "\n" +
                 getHeader( "To: " ) + "\n" +
                   getHeader( "Subject: " ) + "\n" +
                      getHeader( "Date: " ) + "\n\n";
    }

    /**
     * Return a summary of the mail message, suitable for displaying
     * inside a combo box.
     */
    public String shortSummary() {
        return truncatedHeader( "Date: ", 16 ) + " | " +
                   truncatedHeader( "From: ", 25 ) + " | " +
                       truncatedHeader( "Subject: ", 33 );
    }

    /**
     * Return a pretty version of the mail message including the short
     * headers and the message content.
     */
    public String toString( ) {
        return shortHeaders() + "\n" + contents.toString();
    }
}

-------------------------

import squint.*;

/**
 * Provides some standard POP server functions like
 * 1.  Logging in to the pop server
 * 2.  Grabbing Messages
 * 3.  Checking how many messages are available
 * 4.  Logging out of the pop server
 */
public class POPConnection {

    private final int POP_PORT = 110;

    /**
     * The connection to the POP server
     */
    private NetConnection toServer;

    public POPConnection( ) {

    }
```

```
    /**
     * Login to 'server' with 'id' and 'password'.  Return true if and only if
     * the login is successful.
     */
    public boolean login(String server, String id, String password) {
        toServer = new NetConnection( server, POP_PORT );
        toServer.in.nextLine();
        toServer.out.println( "USER " + id );
        toServer.in.nextLine();
        toServer.out.println( "PASS " + password );
        if ( ! toServer.in.nextLine().startsWith( "+OK" ) ) {
            close();
            return false;
        } else {
            return true;
        }
    }

    /**
     * Grab the 'messageNum' mail message from the server.
     * Return null if the message does not exist.
     */
    public MailMessage getMessage( int messageNum ) {
        toServer.out.println( "RETR " + messageNum );
        if ( toServer.in.nextLine().startsWith( "+OK" ) ) {
            String response = toServer.in.nextLine();
            MailMessage result = new MailMessage();
            while ( ! response.equals( "." ) ) {
                result.append( response );
                response = toServer.in.nextLine();
            }
            return result;
        } else {
            return null;
        }
    }

    /**
     * Return the number of available message from the server.
     */
    public int messagesAvailable() {
        toServer.out.println( "STAT" );
        String response = toServer.in.nextLine();
        if ( response.startsWith( "+OK" ) ) {
            String num = response.substring( "+OK ".length() );
            num = num.substring( 0, num.indexOf( " " ) );
            return Integer.parseInt( num );
        }
        return -1;
    }

    /**
     * log out of the POP Server
     */
    public void close() {
        toServer.out.println( "QUIT" );
        toServer.in.nextLine();
        toServer.close();
    }

}
```