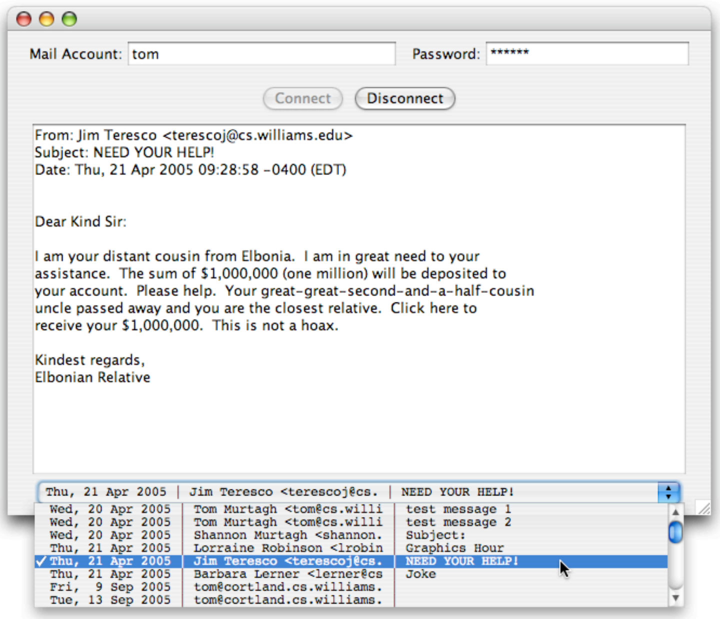
Lab 6

Recursive Revision

Due October 27/28 9PM

This week, we want you to implement 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.



```
From: Jim Teresco <terescoj@cs.williams.edu>
Subject: NEED YOUR HELP!
Date: Thu, 21 Apr 2005 09:28:58 -0400 (EDT)


Dear Kind Sir:

I am your distant cousin from Elbonia.  I am in great need to your
assistance.  The sum of $1,000,000 (one million) will be deposited to
your account.  Please help.  Your great-great-second-and-a-half-cousin
uncle passed away and you are the closest relative.  Click here to
receive your $1,000,000.  This is not a hoax.

Kindest regards,
Elbonian Relative
```

```
 Thu, 21 Apr 2005 | Jim Teresco <terescoj@cs. | NEED YOUR HELP!
  Wed, 20 Apr 2005 | Tom Murtagh <tom@cs.willi | test message 1
  Wed, 20 Apr 2005 | Tom Murtagh <tom@cs.willi | test message 2
  Wed, 20 Apr 2005 | Shannon Murtagh <shannon. | Subject:
  Thu, 21 Apr 2005 | Lorraine Robinson <lrobin | Graphics Hour
✓ Thu, 21 Apr 2005 | Jim Teresco <terescoj@cs. | NEED YOUR HELP!
  Thu, 21 Apr 2005 | Barbara Lerner <lerner@cs | Joke
  Fri,  9 Sep 2005 | tom@cortland.cs.williams. |
  Tue, 13 Sep 2005 | tom@cortland.cs.williams. |
```

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`.

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 behaves externally, we want you to change how it is implemented internally. In particular, we want you to change it so that it uses recursive structures.

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. We also encourage you to work together in pairs on this week's lab to make it even easier to complete.

Of course, there is another catch. 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. It also contains a class named `TimeTrials` that will be used to experimentally measure the efficiency of some of your code.

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`**

  This method handles the process of logging in and out from the 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 sending "USER" and "PASS" commands to the server, the code we have provided depends on a separate class named `POPConnection` to perform these steps. In addition, after the login is complete, the code in our `buttonClicked` method 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.

- **`menuItemSelected`**

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

- **`textEntered`**

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

POPConnection

The `POPConnection` class provides methods that can be used to perform basic interactions with a POP server. It is very similar to the `POPConnection` class presented in class. You can construct a new `POPConnection` with a command of the form

```
POPConnection toServer = new POPConnection( server, userid, password);
```

in which all three parameters are `String`s. Once you have constructed a `POPConnection` it can be accessed using any of the following methods:

- **`public boolean isConnected()`**

  This method returns a boolean value indicating whether the login performed when the connection was constructed succeeded.

- **`public int messagesAvailable()`**

  This method returns the number of messages currently stored on the account.

- **`public MailMessage getMessage( int messageNumber )`**

  This method takes a message number as a parameter and returns the requested message. The message returned is an object of the `MailMessage` class described below.

- **`public void close()`**

  The close method should be invoked to log out from a POP server.

In addition, the `POPConnection` class includes a private `fetchMessage` method that actually contacts the server to retrieve messages for the `getMessage` method.

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. It provides an `addLine` method that can be use to add lines to the `MailMessage` as they are received from the server.

Unlike the `MailMessage` class described in class, this `MailMessage` class stores the message as two separate `Strings`. 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 `addLine` method, two methods can be used to access the contents of the message.

- **`public String toString()`**
  This method returns the entire contents of the body of the method preceded by its most important header lines (To, From, Date, and Subject).
- **`public String shortSummary()`**
  This method returns a single line containing parts of the header 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. Your `StringList` class should include two constructors and two methods:

- **`public StringList()`**
- **`public StringList(String last, StringList rest)`**
  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 `StringList` that includes the new line in addition to all the lines in its `StringList` parameter.
- **`public String toString()`**
  The `toString` 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.
- **`public String getLineStartingWith( String prefix )`**
  This method should take a `String` as a parameter and return a line from the `StringList` that starts with the specified prefix. This method should return the empty string (`""`) if no match is found.

Once you have defined the `StringList` class, we want you to change the declarations of the two variables in the `MailMessage` class 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 `addLine` method and the `getHeader` method.

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 provide two constructors and one method:

- **`public MessageList()`**
- **`public MessageList( MailMessage message, int messageNum, MessageList rest )`**

  The first constructor will take no parameters and return an empty `MessageList`. The other constructor will expect three parameters: a `MailMessage`, 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.

- **`public MailMessage get( int messageNum)`**

  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 local "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 message 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`. Then, modify our `fetchMessage` method so that it adds a pair to the `MessageList` cache associating the message it was asked to fetch with its message number. Finally, modify the `getMessage` method to use the `get` method to see if the message it was asked to retrieve is already in the cache before invoking `fetchMessage`. If the requested message is in the cache, `getMessage` should just return it without invoking `fetchMessage`. If not, it will invoke `fetchMessage` to access the message.

In the context of the `POPClient` we have provided, these changes 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. In fact, once the menu is built, you could actually close the connection to the server.

Getting Started

To start this lab, you should download a copy of the menu POP client described above.

- Launch Safari (you can use another browser, but these instructions are specific to Safari) and go to the "Labs" section of the CS 134 web site (http://www.cs.williams.edu/~cs134).
- Find the link that indicates it can be used to download the `Lab6Starter` program.
- Point at the link. Hold down the control key and depress the mouse button to make a menu appear.
- Find and select the "Download Linked File As ..." item in the menu.

- Using the dialog box that appears, navigate to your "`Documents`" folder and save the `Lab6Starter.zip` file in that folder.
- Return to the Finder, locate the `Lab6Starter.zip` file in your Documents folder, and double-click on it to create a `Lab6Starter` folder.
- Rename the folder using a name including "Lab6" and your name (e.g., FloydLab6). Remember not to include any blanks in the new folder's name.
- Launch BlueJ and use the "Open Project" item in the "File" menu to open your Lab 6 project.

Implementation Plan

A Timing Experiment

1. First, we want you to run a simple timing experiment using the code we have provided. A handout guiding you through this experiment will be provided at the beginning of the lab period. Copies of the timing experiment handout can also be downloaded from the Labs page of the course website.

Define and test StringList

2. 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`.
3. Add definitions for the `lineStartingWith` method and the `toString` method.
4. Test these constructors and methods using the mechanisms explained in the "Working with Multiple Classes" appendix included with last week's lab:
   a) First, you will have to create an empty list. Select the empty list constructor from the menu that appears when you point at the `StringList` icon in the project window, depressing the control key and the mouse button. Type the name "emptylist" into the dialog box that appears and click OK.
   b) Next, create some non-empty `StringList`s. You will do this by selecting the `StringList` constructor that expects two parameters from the menu that appears when you point at the icon for the `StringList` class. This time, the dialog box that appears will require that you type in two parameter values in addition to letting you type in a name for the object you create. For the first parameter, you should just type in a quoted string. For the second parameter, type in the name of one of the `StringList`s you have already created. For the first non-empty `StringList`, this will have to be the name "emptylist". In general, you will use the name of the last list you created. Creating a total of 3 non-empty lists should be sufficient.
   c) Now, test your `lineStartingWith` and `toString` methods on the lists you have created by depressing the mouse on the red icons for the objects in the BlueJ menu and selecting the method names from the menus that appear.

Modify MailMessage to use StringList

5. Modify the `MailMesage` class so that the `contents` and `headers` member variables have type `StringList` instead of `String`. Then, modify the constructor, the `addLine` method and the `getHeader` method so that the `MailMessage` class behaves as before. Test the `POPClient` class to verify that it works as it did before your modifications.

Another Timing Experiment

6. Repeat the timing experiment that we had you perform to evaluate the performance of the `MailMessage` class to see how the change in its implementation impacts the time required to create `MailMessage`s.

Define MessageList

7. 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`. Also define the `get` method.

Modify POPConnection to use a MessageList as a local cache

8. 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.
9. Add code in the `fetchMessage` method to add each message fetched to the `MessageList` cache.
10. Modify `getMessage` so that it uses `get` to see if the desired message is already in the cache before invoking `fetchMessage`. If the message is in the cache, simply return it. If not, use `fetchMessage` to access the message and return the result.
11. Test the program. At this point, the internal structure 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

This labs will be graded on the following scale:

++	An absolutely fantastic submission of the sort that will only come along a few times during the semester.
+	A submission that exceeds our standard expectation for the assignment. The program must reflect additional work beyond the requirements or get the job done in a particularly elegant way.
✓+	A submission that satisfies all the requirements for the assignment --- a job well done.
✓	A submission that meets the requirements for the assignment, possibly with a few small problems.
✓-	A submission that has problems serious enough to fall short of the requirements for the assignment.
-	A submission that is significantly incomplete, but nonetheless shows some effort and understanding.
--	A submission that shows little effort and does not represent passing work.

Completeness / Correctness

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

Style

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

```java
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 {
    // Port number used to contact a POP server
    private final int POP_PORT = 110;

    // The underlying NetConnection to the POP server
    private NetConnection toPopServer;

    // Whether password was accepted
    private boolean connected;

    // Constructor for objects of class POPConnection
    public POPConnection(String server,
                         String user, String password) {
        // Connect to the server
        toPopServer = new NetConnection(server, POP_PORT);

        String response = toPopServer.in.nextLine();

        // Send account information
        toPopServer.out.println("USER " + user);
        response = toPopServer.in.nextLine();
        toPopServer.out.println("PASS " + password);

        // Check that the login was accepted by the server
        response = toPopServer.in.nextLine();

        if (response.startsWith("+OK")) {
            // Determine how many message are on the server
            connected = true;
        } else { // login attempt was rejected
            connected = false;
            toPopServer.close();
        }
    }

    // Determine whether connection was established
    public boolean isConnected() {
        return connected;
    }

    // Returns message number messageNum or  null
    public MailMessage getMessage( int messageNum ) {

        /////////////////////////////////////////////////////
        // ADD CODE HERE TO TRY ACCESSING THE MESSAGE FROM
        // YOUR MESSAGELIST BEFORE ACTUALLY CONTACTING THE
        // SERVER TO GET THE MESSAGE.
        ////////////////////////////////////////////////////

        MailMessage result = fetchMessage( messageNum );

        return result;
    }

    // Retrieve message number messageNum the server or return null
    private MailMessage fetchMessage(int messageNum) {
        toPopServer.out.println("RETR " + messageNum);

        String retrResponse = toPopServer.in.nextLine();

        if (retrResponse.startsWith("+OK")) {
            MailMessage curMessage = new MailMessage();
            retrResponse = toPopServer.in.nextLine();

            while (!retrResponse.equals(".")) {
                curMessage.addLine(retrResponse);
                retrResponse = toPopServer.in.nextLine();
            }

            return curMessage;
        } else {
            return null;
        }
    }

    // Retrieve the number of messages available on the server
    public int messagesAvailable() {
        toPopServer.out.println("STAT");

        String statInfo = toPopServer.in.nextLine();
        statInfo = statInfo.substring(4);

        int countEnd = statInfo.indexOf(" ");

        return Integer.parseInt(statInfo.substring(0, countEnd));
    }

    // Terminate the connection
    public void close() {
        toPopServer.out.println("QUIT");
        String response = toPopServer.in.nextLine();
        toPopServer.close();
    }
}
```

```java
/**
 * This class represents an email message. It provides methods
 * to retrieve the full header, a short header, a message summary,
 * and an abridged view of the  message.
 */
public class MailMessage {

    // Widths of fields present in short message summaries
    private final int DATE_WIDTH = 16;
    private final int FROM_WIDTH = 25;
    private final int SUBJECT_WIDTH = 33;

    // The headers and the body of the mail message
    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 addLine( String newLine ) {
        if ( contents == null ) {
            if ( newLine.equals("") ) {
                contents = "";
            } 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";
    }

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

    /**
     * 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.*;
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 program's window
    private final int WINDOW_WIDTH = 650, WINDOW_HEIGHT = 470;

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

    // 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 & password
    private JTextField user = new JTextField( 20 );
    private JTextField pass = new JPasswordField( 15 );

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

    // Menu used to select messages;
    private JComboBox messageSelector;
    private Font menuFont = new Font( "Courier", Font.PLAIN, 12 );

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

    // Our we currently connected to the POP server
    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
    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 );

            // Install buttons used to log in and out
    curPanel = new JPanel();
    curPanel.add( login );
    curPanel.add( logout );
    contentPane.add( curPanel );

    // Install the text area used to display messages
    message.setEditable( false );
    contentPane.add( new JScrollPane( message ) );
}
```

```
/*
 * When the button is clicked, interact with the POP server to
 * log in our out.
 */
public void buttonClicked( JButton which ) {

    if ( which == login ) {
        // Connect to the server
        toServer = new POPConnection( SERVER, user.getText(),
                                      pass.getText());
        if ( toServer.isConnected() ) {
            // Create menu of all messages available in mailbox
            messageSelector = new JComboBox();
            messageSelector.setFont( menuFont );

            // Fill the menu with message summaries
            int messNum = 1;
            int totalMessages = toServer.messagesAvailable();
            while ( messNum <= totalMessages ) {
                MailMessage mess = toServer.getMessage(messNum);
                messageSelector.addItem(
                         messNum + ") " + mess.shortSummary() );
                messNum = messNum + 1;
            }                                                      }
            contentPane.add( messageSelector );
        } else {
            message.setText(
                      "Unable to login. Check your password." );
        }

    } else if ( which == logout ) {
        connected = false;

        // Terminate the connection and remove the message menu
        toServer.close();
        message.setText( "" );
        contentPane.remove( messageSelector );
    }

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

    // Ensure that updated menu is displayed
    repaint();
}
```

```
/**
 * Request the selected message from the server
 * and display it in the text area.
 **/
public void menuItemSelected() {
    MailMessage 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();
    }
```