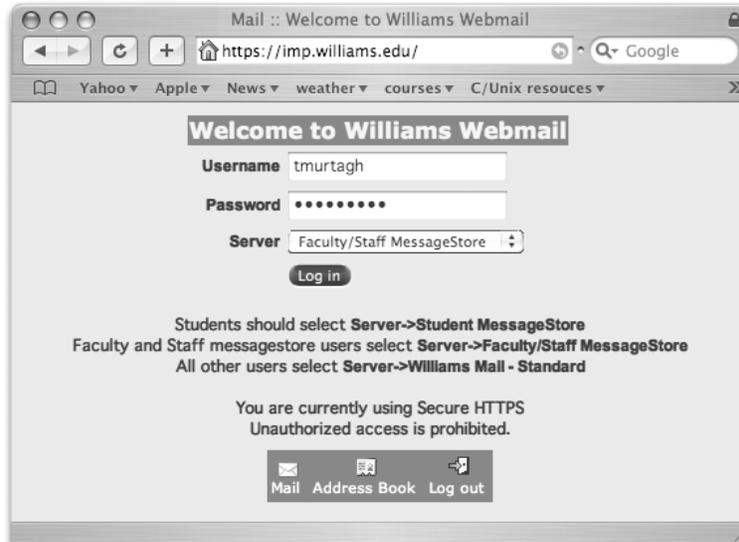


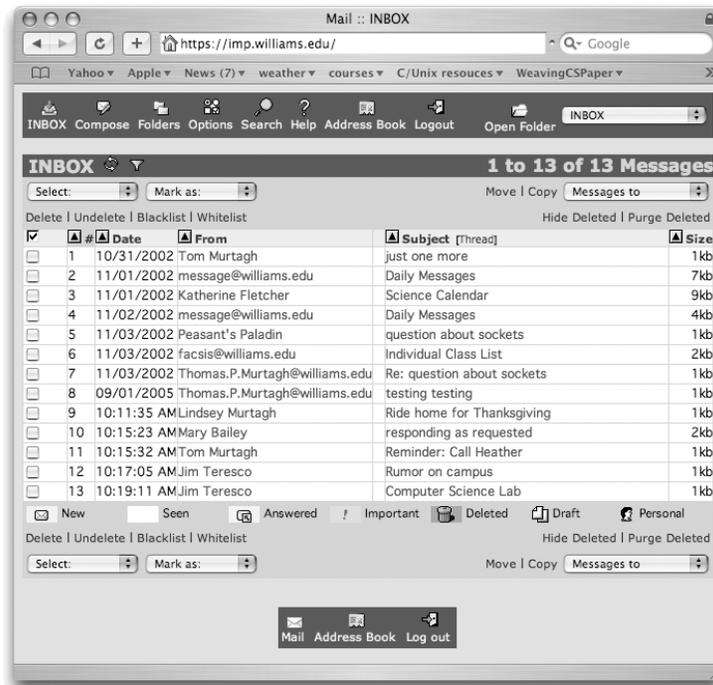
Lab 10: Webmail

Due: 11PM, April 24/25

Chances are, most of you are familiar with the window shown below. It is a sample of the interface provided to users logging in to access their mail using the Williams College webmail service.



After completing this login page correctly, a user is presented with a screen like the one shown below through which the user can read or send email messages.



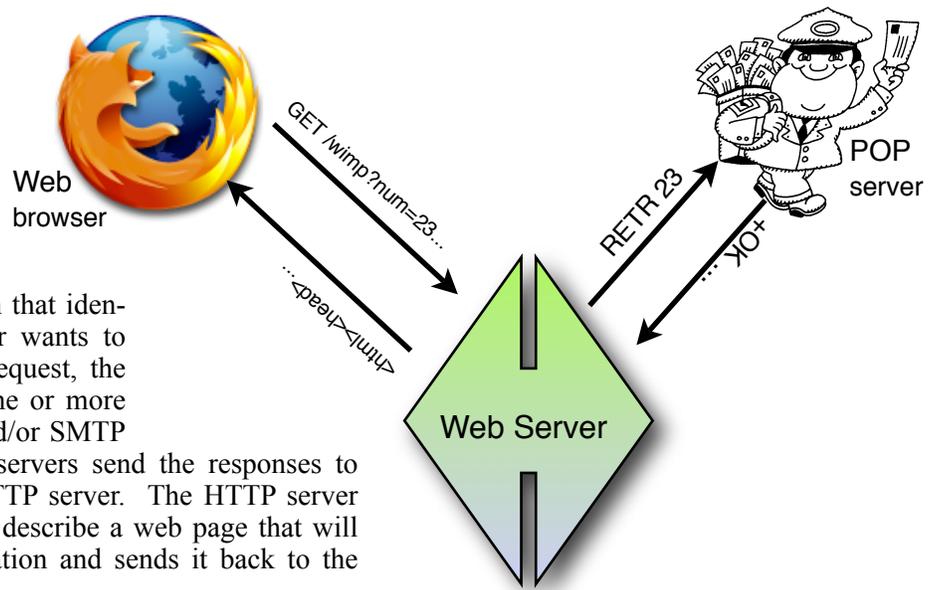
Despite the fact that these windows are very familiar to you, if you think about their contents in the context of this course, you should realize there is something odd going on. You have learned how the Internet's email system works. Email client programs provide access to email by using email protocols like

POP, IMAP, and SMTP to communicate with email servers. On the other hand, web client programs (i.e., browsers) provide access to web pages by using the HTTP protocol to communicate with web servers. How can you read your mail using a web client that depends on HTTP when mail access requires the use of POP or IMAP?

HTTP is a protocol that provides rules by which a client can request that a server send it some data in HTML format. Often, an HTTP server responds to the requests it receives by simply sending back the contents of files of HTML stored on the server, but this does not have to be the case. The server can instead contact other servers using other protocols to collect the desired information, encode the information collected in HTML and then send this HTML back to the client. In fact, this is exactly how a web-based mail server like the one provided by OIT is implemented.

The diagram on the right suggests how this works. First, based on which link or button a user clicks, the web browser sends an HTML “GET” request to the HTTP server program designed to provide mail access.

The URL sent in this request will include information that identifies the message(s) the user wants to see. When it receives this request, the HTTP server in turn sends one or more requests to the user’s POP and/or SMTP server. The POP or SMTP servers send the responses to these requests back to the HTTP server. The HTTP server then constructs the HTML to describe a web page that will display the requested information and sends it back to the web browser to be displayed.



For this assignment, we would like you to implement an HTTP server that functions in this way. Like the very first Java programs you wrote for this course, your HTTP server will only provide half of the functionality expected of a mail server. It will enable users to view their mail messages, but not to send mail. That is, it will provide access to a POP server, but not to an SMTP server.

We will refer to the program we want you to write as WIMP (Williams Internet Mail Processor) in honor of the program that OIT actually uses to provide webmail service. That program is named IMP. If you hunt around using Google a bit, you will discover that a) IMP is widely used by many institutions beyond Williams, and that b) the creator of IMP is a former Williams CS major, Chuck Hagenbuch '00, who developed the first version of the program while still at Williams.

The WIMP HTTP Interface

Your webmail server should respond to HTTP requests of two forms. When users first connect to your server, they will enter a URL of the form:

```
http://your.machine.net:8080/wimp
```

Your program will then receive an HTTP request of the form

```
GET /wimp HTTP/1.1
```

Your program should respond to such requests by sending back the HTML for the page shown on the right.

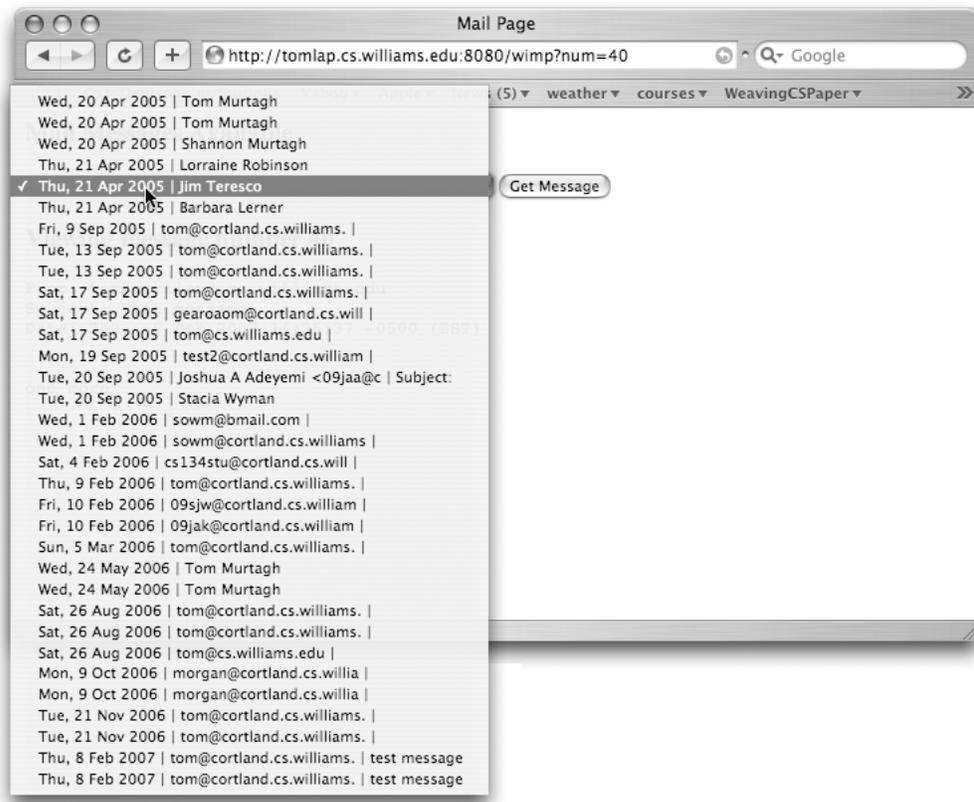
The bottom portion of the window will display the text of a single email message. When a user first connects, the last message received on the account should be displayed. Above the contents of the message there will be a menu containing descriptions of all the messages within the user's account.

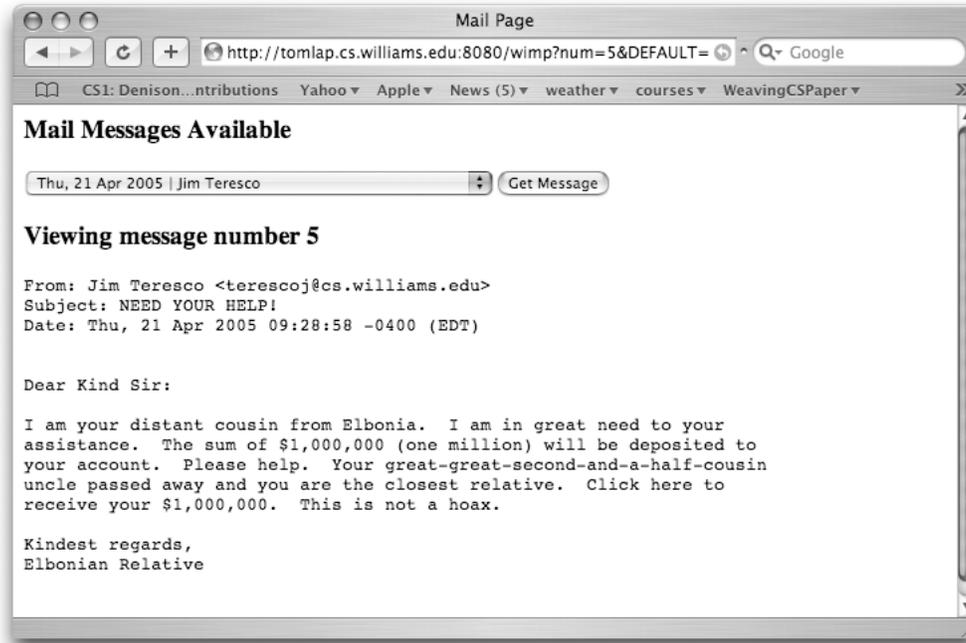


Selecting one of these menu items as shown below and then pressing the “Get Message” button next to the menu will cause the browser to send a request of the form

```
GET /wimp?num=nn HTTP/1.1
```

to your server where “nn” is replaced by the number of the menu item that was selected when the button was clicked. This will cause your program to send HTML describing a page including both the menu of all of the user's message and the contents of the requested message as shown on the next page.



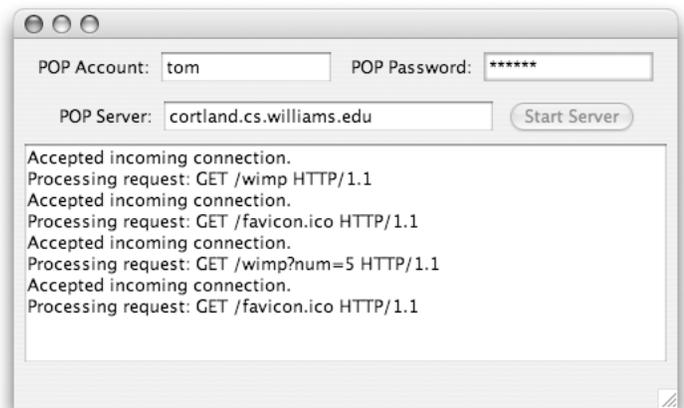


The menu displayed in the page should be pre-scrolled so that the item selected in the menu is the item describing the message displayed within the window. The user should then be able to display additional messages by first selecting the desired message from the menu and then pressing the “Get Message” button.

If your server receives a request that does not start with “/wimp” it should respond by sending the client HTML that will display a web page reporting an error.

The WIMP GUI Interface

The most interesting aspects of your program for this week will be the way in which it responds to incoming HTTP requests. It will, however, also provide a very simple GUI interface to enable you to monitor its behavior and to control which POP server and account it uses when responding to HTTP requests. A sample of what this GUI interface might look like is shown on the right.



Across the top of the program’s window there are `JTextField`s and a `JPasswordField` that are used to describe the POP account that should be used to fetch mail for the user and a `JButton` used to start the server. Your program will not respond to incoming HTTP requests when it is first run. Instead, after running the program, you will have to fill in the fields describing the mail account that should be used and then press the “Start Server” button. At this point, your program will create a `TCP` port listening for connections on some relatively large port number (8080 is my favorite). It will then be ready to respond to HTTP requests.

Below the text fields and button is a `JTextArea` which you can use to log information about what your program is doing. In the version of the program used to create the window above, we displayed a line each time a connection was established and each time an HTTP request was received.¹ You can display this information or any other information you find useful when debugging your code.

As this interface suggests, your program is intended to function as a very personal web mail server. Unlike the IMP server run by OIT, it will not provide mail access to multiple users. While it is running it will only provide access to the account whose information was provided through the GUI interface. This is not realistic. In particular, it is not secure. While your program is running anyone clever enough to guess the name of the computer you are using and the magic URL “wimp” will be able to access your mail without knowing your account id or password. We decided not to ask you to implement an HTTP interface that would provide security and/or support multiple users to keep the lab as simple as possible.

Taking Advantage of Your Earlier Work

The menus displayed in the web pages shown before should look very familiar. They look just like the `JComboBoxes` shown in the description of Lab 6. In fact, the functionality provided through this program is very similar to what we asked you to provide in Lab 6. The only difference is that in this week’s lab, you are supposed to write a specialized web server (rather than a mail client) and users are supposed to access this functionality through a web browser (rather than through a client you wrote).

As a result, it should not come as a surprise that you will be able to reuse some of the code you write for Lab 6 to help you complete this lab. In fact, it should be possible for you to use copies of the `POPConnection`, `MessageList`, `MailMessage`, and `StringList` classes you completed six weeks ago as the basis for this week’s lab. If you do this, you will be half done before you even start!

If it helps, you may modify these classes, but this should not be necessary. In our sample solution to this lab exercise, we used *unmodified* version of the four mail classes from our sample solution to Lab 6.

Basically, when your web server receives a “GET /wimp” request, it should create a new `POPConnection`. It should then invoke the `POPConnection`’s `login` method using the account information entered in the GUI components in the program’s window. If the login succeeds, it should associate this `POPConnection` with an instance variable so that it can use it later to respond to subsequent “GET /wimp?num=nn” requests. Then, it should build a `String` containing the HTML to describe a web page like those shown above containing a menu of message summaries and the text of the last message stored in the account.

You will build this page by using the `POPConnection`’s `messagesAvailable` and `getMessage` methods together with the `shortSummary` method of the `MailMessage` to fetch all of the messages stored in the account and build an array of lines summarizing the messages. This array will then be used to construct the HTML for the web page as described below. Once this is done, you can assume that the `POPConnection` has cached copies of all of the messages stored in the account in its `MessageList`. Therefore, you can close the `POPConnection` so that your POP account does not remain locked while you examine messages in your web browser.

If the login fails, your code to handle a “GET /wimp” request should set the `POPConnection` instance variable to null, and send HTML back to the client to display an error message in the web page that looks something like the window shown below.

¹ Note that from our log you can see that web browsers sometimes send requests you were not expecting. Every time our browser requests a page it also requests a “favicon” image (favorite icon) to display in the address window. This means it is important that your server responds correctly when it receives requests other than those starting with “/wimp” by returning an error page.



When your server receives a “GET /wimp?num=nn” request, it will first check to make sure that the `POPConnection` instance variable is not `null`. If it is `null`, you should respond by sending the HTML for an appropriate error page to the server. If it is not `null`, then it can assume that the `POPConnection` holds cached copies of all of the mail messages in the account. Therefore, even though the account is no longer logged in, you can continue to use the `getMessage` method to access these messages to build the HTML that should be sent back to the client.

Some HTML Basics

We don’t want you to have to spend too much time learning HTML to complete this assignment. With this in mind we have designed the layout for the web pages your program needs to complete so that they can be constructed with only minimal knowledge of HTML. In addition, we will provide several classes in the starter folder that will help you produce the HTML you need to send back to the client.

We have already discussed some of the basics of HTML in class. HTML provides a way to “mark up” plain text to indicate how it should be displayed to a user. HTML provides mark up “tags” to indicate that certain sections of text should be displayed in bold font, centered on the page, displayed as a separate paragraph from surrounding text, etc. Each HTML tag consists of a tag name surrounded by the “angle brackets” `<` and `>`. For example the tag `<p>` is used to indicate the beginning of a new paragraph. Frequently a pair of tags is placed before and after a section of text that is to be displayed in some special way. In this case, the name of the second tag is formed by placing a `/` before the name of the first tag in the pair. For example, to make the phrase “Big words” appear in bold face, you would write

```
<b>Big words</b>
```

There are several HTML tags that appear in the specification of almost every web page that do little more than say “This is a web page.” The first of the classes we will provide is designed to make it trivial to place the text you actually want to display at the appropriate point within these standard tags. This class is named `WebPage`. Its constructor requires no parameters. The class provides a method named `toString`. This method returns a `String` containing the complete HTML to describe an empty web page. For example, if you first create a `WebPage` object by saying

```
WebPage blank = new WebPage( );
```

and then send the text that describes the page through a `NetConnection` to a web browser by executing an invocation like

```
toClient.out.println( blank.toString() );
```

then the web page shown below will appear in the client’s browser window.



Not very exciting! But, think for a moment what would happen if you simply created a new `GUIManager`. It would display an empty window, much like the empty web page. It isn't very exciting to create a new `GUIManager`, but as you have seen throughout this semester, `GUIManager` is an important class because you can extend it to describe programs with interesting GUI interfaces. Similarly, the `WebPage` class is defined to enable you to extend it to describe interesting web pages.

In fact, `GUIManager` and `WebPage` are actually quite similar. If you define a class that extends `WebPage`, you can add various items to the `WebPage` much as you add GUI components to the `ContentPane` of a `GUIManager`. Then, when you invoke the `toString` method of the class, it will produce the HTML to describe a web page containing the items you added.

To make this concrete, consider the error page shown earlier. In the starter folder, we have included a class named `ErrorPage` that extends `WebPage` and is designed to make it quite easy to generate the HTML for such an error page. In particular, if you create an `ErrorPage` by saying:

```
ErrorPage logErr = new ErrorPage( "Unable to log in to POP account" );
```

and then execute the command

```
toClient.out.println( logErr.toString() );
```

the error page shown earlier will be displayed in the client's browser.

The complete code for the `ErrorPage` class is shown below:

```
public class ErrorPage extends WebPage {
    public ErrorPage( String message ) {
        this.setTitle("Error");

        contentPane.add( new WebLabel(message) );
    }
}
```

It uses the two key features of the `WebPage` class. First, the `WebPage` class contains a method named `setTitle`. You can use this method to specify a `String` that should be displayed in the web browser window's title bar.

More importantly, like the `GUIManager` class, the `WebPage` class provides a variable named `ContentPane`. You can add items to the page you are trying to build using the method `add`. The `ErrorPage` class uses this method to add the message passed to its constructor to the page. Recall that we could not

just add `Strings` that we wanted displayed in a `GUIManager` to the `contentPane`. Instead, we had to construct a `JLabel` to hold the `String`. Similarly, to add a `String` to a web page, we construct a `WebLabel` from the `String`.

The `toString` method of any class that extends `WebPage` returns a `String` containing the HTML for the desired page. For example, the HTML sent to the client by executing the commands

```
ErrorPage logErr = new ErrorPage( "Unable to log in to POP account" );
toClient.out.println( logErr.toString() );
```

would be

```
<html><title>Error</title>
<body>
<h3> Unable to log in to POP account </h3>
</body></html>
```

Most of this HTML is a collection of fairly standard HTML tags that say little more than “I am a web page.” The only parts that are specific to this page are the text “Error” in the first line that determines what is displayed in the web browser window’s title bar and the third line that describes the actual contents of the page. This third line is actually produced by the `WebLabel` class.

The `WebLabel` class is also very simple. HTML includes a set of tags for indicating that a piece of text should be displayed as a “header”. The tags used for this are `h1`, `h2`, `h3`, ..., `h7`. The smaller the number, the bigger the header. Thus, the HTML

```
<h3> Unable to log in to POP account </h3>
```

Says that “Unable to log in to POP account” should be displayed as a moderately large header.

Given this tidbit about HTML, the structure of the `WebLabel` class is quite simple. Its code is shown below.

```
public class WebLabel extends WebComponent {
    private String label;

    public WebLabel(String label) {
        this.label = label;
    }

    public void setText( String label ) {
        this.label = label;
    }

    public String toString() {
        return "<h3>" + label + "</h3>";
    }
}
```

Just as the `WebPage` class has a `toString` method that returns the HTML to describe an entire web page, the `WebLabel` class includes the definition of a `toString` method that returns the HTML that describes the part of the web page it is expected to create. It simply put “<h3>” and “</h3>” tags around the text to be displayed. This text can either be provided as a parameter to the `WebLabel` constructor or set using a `setText` method.

Note that the `WebLabel` class extends a class named `WebComponent`. Just as the `ContentPane.add` method of a `GUIManager` will allow you to add any GUI component to a program’s window, the `ContentPane.add` method of a `WebPage` will let you add any `WebComponent` to the page. Therefore, one can define other classes like `WebLabel` by extending `WebComponent`.

In fact, that is just what we have done. In the starter project for this lab, we have included a number of classes you can use to generate the web pages your server should return. These classes are described below. The bad news is that we have not provided all the classes you will need. In particular, as part of this lab, we want you to write one class that extends `WebPage` and one that extends `WebComponent`.

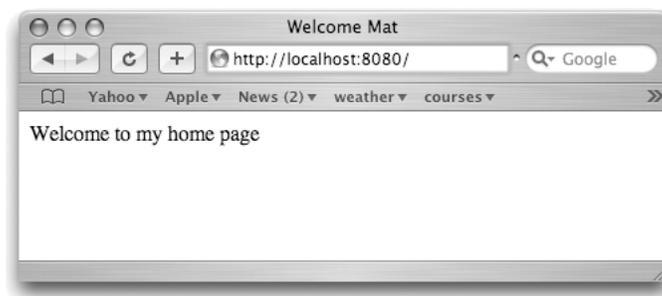
We provide you with the following classes:

WebTextArea

Normally, a browser is free to reformat the text it receives so that each line of text nicely fills the screen. In doing so, it may eliminate blank lines and spaces that appeared in the original HTML. For example, if you sent the HTML

```
<html>
  <head> <title> Welcome Mat </title> </head>
  <body bgcolor=#FFFFFF>
    Welcome to
      my
    home page
  </body>
</html>
```

The web page displayed would look like:



even though the words “Welcome to” appeared on a separate line from the words “my” and “home page” within the HTML.

The `<pre>` tag overrides this behavior. The `<pre>` tag tells the browser that all of the text until the following `</pre>` tag has been “pre-formatted” and that the number of spaces and new-lines should not be changed. Therefore, if you send the following HTML to a browser:

```
<html>
  <head> <title> Welcome Mat </title> </head>
```

```

<body bgcolor=#FFFFFF>
  <pre>
    Welcome to
      my
    home page
  </pre>
</body>
</html>

```

the resulting page will look like:



The `WebTextArea` class is used to describe a section of a web page that will display such pre-formatted text. In particular, you will use the `WebTextArea` class to describe the part of your web pages that displays the requested email message.

You can set the text to be displayed by a `WebTextArea` in two ways. You can either provide the text as a `String` parameter in a `WebTextArea` constructor or as a parameter to an invocation of the `setText` method. For example, if `message` was a `String` variable holding an email message, you could display this message in a `WebPage` by executing code like:

```

WebTextArea display = new WebTextArea();
display.setText( message );
contentPane.add( display );

```

Actually, `WebTextArea` does a bit more than place text between `<pre>` and `</pre>` tags. If there are any `<`'s or `>`'s in the text displayed by a `WebTextPage`, they have to be treated specially or they might confuse the browser into thinking they were parts of HTML tags themselves. Therefore, the `WebTextArea` class replaces all `<`'s with the sequence `"<"`, all `>`'s with the sequence `">"`, and all `&`'s with the sequence `"&"`. These strange sequences are a standard part of the HTML language known as *entities*.

WebButton

We provide a class named `WebButton` that produces HTML to describe buttons like the "Get Message" button that should be included in the pages your server generates. Like the other classes described above, you can provide the label for the button either as a parameter to the constructor or using a `setText` method. For example, the construction

```

new WebButton( "Get Message" )

```

could be used to construct the button you should display.

WebForm

When you click on a button in a web page, the web browser in which the page is displayed has to gather together information from various parts of the page and send it to the server as an HTTP request. For example, when a user clicks on your “Get Message” button, we want the browser to determine the number of the item currently selected in your menu of message summaries and include that number in an HTTP request of the form “GET /wimp?num=nn”. This means that the components of this HTTP request, including the use of the identifiers “wimp” and “num” have to somehow be specified within the HTML your server sends to describe the page.

Your page will contain only two components that will be involved in this process, the menu and the button. You have certainly seen pages with multiple buttons and many, many menus and text fields in which users can enter information. HTML has to provide a way to include tags that indicate which components should be grouped with which buttons when determining what information to include in the HTTP request generated when a button is pressed. The `WebForm` class provides a means through which you can easily generate such HTML tags.

A `WebForm` is used quite a bit like a `JPanel` is used when constructing a GUI interface. It is a `WebComponent`, so it can be added to the `ContentPane` of a `WebPage`. At the same time, it has its own `add` method through which you can add `WebComponents` to a `WebForm`. You can add any `WebComponent` you want to a form, but typically they will include at least one `WebButton` and things like menus through which users can enter additional information.

When you create a `WebForm`, you must include the `WebForm`'s name as a `String` parameter to the constructor. The `WebForm` class includes this name in the HTML it produces in such a way that the name will be included in an HTTP generated by the browser when any `WebButton` included within the `WebForm` is clicked. Thus, when creating the HTML for a page that displays an email message, your program will create a `WebForm` using an instruction like

```
WebForm menuForm = new WebForm( "/wimp" );
```

Then, it will add the menu to this form and also add the button using an instruction like:

```
menuForm.add( new WebButton( "Get Message" ) );
```

Finally, it can add the completed form to the web page by saying

```
contentPage.add( menuForm );
```

The HTML produced when the `toString` method of a `WebForm` is invoked is actually quite simple. It begins with a tag of the form:

```
<form action="/wimp">
```

This will be followed with the tags to describe all of the component added to the form. Finally, the tag “</form>” will be placed at the end.

WebPane

The final class we include in the starter project is named `WebPane`. Like a `JPanel`, a `WebPane` is a general purpose container for holding components. You will not directly use this class in your program, but our definitions of the `WebPage` and `WebForm` classes both depend on `WebPane`.

WebComboBox

To complete this program, you will need to define one additional `WebComponent` to create the HTML for the menu displayed by your program. To help you understand what this class must do, a sample of the HTML for a shortened version of the menu shown in the example above is displayed below:

```
<select name="num">
<option value=1> Wed, 20 Apr 2005 | Tom Murtagh <tom@cs.willi | test message 1 </option>
<option value=2> Wed, 20 Apr 2005 | Tom Murtagh <tom@cs.willi | test message 2 </option>
<option value=3> Wed, 20 Apr 2005 | Shannon Murtagh <shannon. | Subject: </option>
<option value=4> Thu, 21 Apr 2005 | Lorraine Robinson <lrobin | Graphics Hour </option>
<option value=5 SELECTED> Thu, 21 Apr 2005 | Jim Teresco | NEED YOUR HELP! </option>
<option value=6> Thu, 21 Apr 2005 | Barbara Lerner <lerner@cs | Joke </option>
<option value=7> Fri, 9 Sep 2005 | tom@cortland.cs.williams. | </option>
<option value=8> Tue, 13 Sep 2005 | tom@cortland.cs.williams. | </option>
<option value=9> Tue, 13 Sep 2005 | tom@cortland.cs.williams. | </option>
</select>
```

The HTML begins with the tag

```
<select name="num">
```

and ends with a “`</select>`”. In between there is a list of items of the form:

```
<option value=nn> ... menu item text ... </option>
```

describing individual menu items. One of these items may include the word “selected” in the opening `<option ...>` tag. This item will then be displayed as the initially selected item in the menu.

The specification `name="num"` in the `<select ...>` tag determines how the item selected in the menu when the form’s button is clicked will be included in the HTTP request generated. Whatever string appears after “`name=`” will be paired with the number of the menu item selected. For example, if we replaced the `<select ...>` tag shown above with

```
<select name="menuItem">
```

then an HTTP request sent when the button was pressed might look like

```
GET /wimp?menuItem=8 HTTP/1.1
```

The constructor of the `WebComboBox` class you define should take at least one parameter. This parameter will be a `String` that should be included as the name of the menu in its `<select ...>` tag. You may want to also define the constructor so that it accepts an array of `Strings` as a second parameter. Each of the `Strings` in the array should then be included as an item in the menu. You should decide whether to define a constructor with one or two parameters by choosing the approach that will make it easiest for you to create and update the web pages your program must generate.

Similarly, the class may choose to provide either two or three public methods. There should definitely be a `toString` method. This is where you will actually do the work of generating the HTML. There should also definitely be a method named `setSelectedItem` that takes an integer value as a parameter and determine which menu item will be marked as “selected” in its `<option ...>` tag. You may also want to define a `setItems` method which takes a new array of `Strings` that should appear as the items in the menu.

MessagesPage

The second HTML related class you should define is named `MessagesPage`. It should extend the `WebPage` class. It should add four `WebComponents` to its content pane: two `WebLabels`, a `WebForm` to hold the message menu and “Get Message” button, and a `WebTextArea` to display the requested email message. The constructor of your `MessagesPage` class should take an array of message summary `Strings` as a parameter. The class should include a `setMessage` method that takes two parameters: the number of the message to be displayed and an `EmailMessage` containing the text of that message. You will not define a `toString` method in this class. Instead, it will inherit `toString` from the `WebPage` class. This inherited method will return the HTML for all of the components you add to the `contentPane` within your code.

Dealing with HTTP

Your program’s main class will extend `GUIManager` as usual. Its constructor will create the GUI interface through which the user will enter POP account information. It will include a `buttonClicked` method that creates a `TCPPort` (and disables the “Start” button). Finally, it will have a `connectionEstablished` method that handles individual HTTP requests.

Within the `connectionEstablished` method you will have to use `String` methods to dissect the HTTP requests received and construct and send the HTML for web pages in response. Remember to use `private` methods within your main class to keep this code well structured if it begins to get at all complicated.

Getting Started

Download the starter file `Lab10Starter.zip` from the course website <http://www.cs.williams.edu/~cs134/s07/labs> and unpack it in your `Documents` folder. Rename the folder that is created so that it contains your name and “Lab10”. Open the project with BlueJ.

We also want you to import all of your classes (except `POPClient`) from Lab 6. Under the BlueJ “Edit” menu there is an option to “Add Class From File.” Selecting this option brings up a file chooser. If you double click on your Lab 6 folder within this file chooser, you’ll find a list of project files. Among these files are `.java` files corresponding to your classes. If you hold down the command key (this is the key with the clover or open apple on it) you can select multiple `.java` files at the same time. Select all the `.java` files except the `POPClient.java` file. Then click “Add.”

Debugging Tips

Don’t forget to close the `NetConnection` your server accepts from each client after you finish sending the HTML describing the web page the client should display. If not, the client will wait patiently to see if you are going to send it more HTML before displaying anything. The web browser will appear to be loading the next page for a very long time.

If the browser doesn’t display the page quite the way you wanted it to look, it might help to be able to look at the HTML that was actually sent rather than just the result produced when the browser tried to interpret that HTML. Most browsers provide a “View Source” option. For example, there is a “View Source” menu item in the Safari “View” menu. Selecting this item will pop up a new window displaying the text of the HTML the browser received from your server.

Submission instructions

As usual, make sure you include your name and lab section in a comment in each class definition. Find the folder for your project. Its names should be something like `FloydLab10`.

- 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”.

You can submit your work up to 11 p.m. two days after your lab (11 p.m. Wednesday for those in the Monday Lab, and 11 p.m. Thursday 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.

APPENDIX: Review of TCPPort Features

- You can create a `TCPPort` using a construction of the form

```
new TCPPort( portNum )
```

where `portNum` is the number you want associated with your program’s port.

- An invocation of the form

```
somePort.addConnectionListener( this );
```

can be used to request that the `TCPPort` notify your program whenever a new connection is requested by a client. When a connection is made, the `TCPPort` will execute any code you place in a method defined with a heading of the form

```
public void connectionEstablished( TCPPort whichPort ) {
```

- An invocation of the form

```
somePort.acceptNetConnection( )
```

will cause your program to wait until a client requests a connection. The invocation will then return a `NetConnection` between your program and the new client.