Test Program

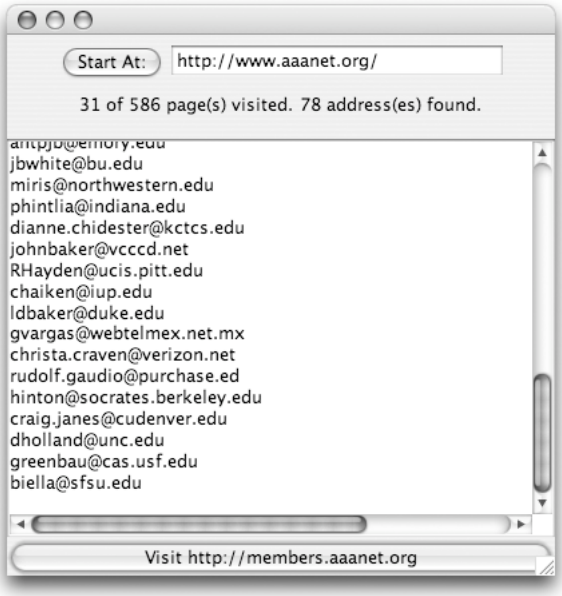
Lovely spam! Wonderful spam!

Due: 5PM, December 2, 2005

A test program is a laboratory that you must complete on your own, without the help of others. It is a form of take-home exam. You may consult your text, your notes, your lab work, or our on-line examples and web pages, but use of any other source for code is forbidden. You may not discuss these problems with anyone aside from the course instructors. You may only ask the TAs for help with hardware problems or difficulties in retrieving your program from a disk or network.

Your goal for this assignment will be to construct a program that would be of value to anyone in the business of sending spam. Your program will not actually send spam. It will just build a collection of email addresses by scanning web pages and extracting any email addresses it can find in those pages.

A snapshot of what your program's interface might look like while it is running is shown below.



The text field near the top of the window is used to enter the URL of a web site from which the user would like the program to start searching. After entering an address in this field, the user should press the "Start At:" button. The program will then fetch the text of the HTML file for the page and search through the text looking for both email addresses and URLs. Internally, the program will keep collections of all the email addresses and all the URLs it has found since starting a search. As soon as it finishes searching one page, the program will display the URL of another page in the button at the bottom of its window. When the user clicks on this button, the program will visit the page whose URL appears in the button, search the page for additional URLs and email addresses, and add the addresses and URLs found to its collections. The program displays its entire collection of email addresses in the text area in the center of the window. It displays the current size of its collections of addresses and URLs and the total number of pages it has searched above the text area.

Obviously, if you were a real spammer, you would remove the button at the bottom of the screen and just write a loop to find as many addresses as you want once the user presses "Start At:". We have included the button in our design because it will make your program easier to debug and will prevent any of you from accidentally flooding the campus network with thousands of requests for web pages in a matter of seconds.

Getting started

To guide you in implementing this program, we will provide descriptions of the classes and methods you should implement below. While we have provided such descriptions to guide you in other labs, for this assignment, these descriptions are intended to serve an additional role. Since you are expected to work on this assignment alone, we recognize that some of you may not be able to complete all components of the program successfully. In such situations, we want you to have the reassurance of knowing that you have completed significant portions of the assignment correctly. With this in mind, we will provide a starter project containing classes that will enable you to independently test each of the classes we want you define.

You will find a copy of the starter folder for this assignment in a Zip file on Cortland in the Lab Materials folder. Drag this file to your documents folder and double-click on it to unpack the contents to form a new folder. Change the folder's name to something like "TestProgramJones".

Step 1: An HTTP client class

The first class we want you to define will be named `HTTPConnector`. It is intended to provide a simple interface through which a program can request the text describing a web page from a web server. In fact, the interface provided by this class will be quite simple. Its constructor will expect no parameters (and do nothing!). It will include just one method named `getWebFile` which will take the URL of a web site as a parameter and return the text received from the server as a `String`.

To implement `getWebFile`, you need to know a bit about the rules of the HTTP protocol. Fortunately, the details you need to know are quite a bit simpler than other protocols we have studied like POP, TOC, and SMTP.

If a program wants to retrieve the text describing a page with the URL

```
http://www.someserver.com/somedirectory/somefile.html
```

it should perform the following steps:

1. Create a `NetConnection` to the HTTP port (port 80) on `www.someserver.com`.
2. Send the following two lines to the server:

   ```
   GET /somedirectory/somefile.html HTTP/1.0
   host: www.someserver.com
   ```

3. Send an empty line to the server
4. Get the the string sent as a response by the server.
5. Close the connection.

Performing these steps requires some manipulation of the URL provided. In particular, you have to extract the name of the web server and the path to the desired file. You should be able to imagine how to do this using the String `indexOf` and `substring` methods. There is, however, an easier way.

The Java libraries include a class named `URL`.[1] Given a String, you can turn it into a `URL` using a construction of the form

```
new URL( "http://www.someserver.com/somedirectory/somefile.html" )
```

The `URL` class includes two accessor methods named `getHost` and `getPath` that will return the server name and file path components of the URL as `String` values. Working with these two methods is quite a bit easier than extracting the server name and path from a `String`. As a result, we want you to define your `getWebFile` method to take a `URL` as a parameter rather than a `String`.

When you think you have completed this class, you can test it by running the `ConnectorTester` class provided in the starter folder. When you create a new `ConnectorTester`, a window will appear containing a text field in which you can type the URL of a web page. If you press "Load Page" after entering this URL, the web page requested should be displayed (in a somewhat rough form) in the body of the window if your `HTTPConnector` class is functioning correctly.

Step 2: Simple `String` lists

Your completed program will have to manage two lists: A list of all the email address it has found and a list of the URLs it has found. While it is sometimes convenient to represent web addresses using the `URL` class, it is also possible to represent a URL as a `String`. As a result, we can provide the tools needed to manage both lists by defining a class named `StringList` to implement a list of `String`s.

The complete program will need to be able to perform four operations on string lists.

1. Obviously, when it finds a new email address or URL, your program will need a way to add it to a list. Therefore, your `StringList` class should include an `add` method that takes the string to be added as a parameter. If the `String` passed to the `add` method is already in the list, the add method should do nothing. We don't want duplicates in our lists. You may want to define a private `contains` method to make writing the `add` method easier.

2. To make it easy to display information about how many web pages and email addresses have been found, your StringList class should include a `size` method that returns the number of items in the list.

3. The complete program will need a way to get a URL that has not been previously visited from the list of URLs. With this in mind, define a `getNext` method that will return one of the `String`s in the list. If all of the members of the list have already been returned, then `getNext` should return `null`.

4. Finally, to display all the email addresses found, your implementation of a list of Strings should provide a method that will return the entire list as a single String with the individual members separated by new lines. Define a `toString` method for this purpose.

You could implement the StringList class either as a recursive class or using an array. We want you to use an array of `String`s. This means that the size of the list will be limited by the size of the array. To provide flexibility, the constructor for `StringList` should take the size of the array to use as a parameter. If the array becomes full, the `add` method should just ignore requests to add additional `String`s to the list.

When you believe you have completed the implementation of `StringList`, you can use the `StringListTester` class included in the starter folder to test its correctness. When you create a new

---

[1] While Java's standard library does include a class named URL, we have actually included our own implementation of URL in the starter folder. The built-in URL constructor throws an exception that would require you to write a try-catch statement as we did when using the ImageIO class. Our version of URL eliminates this need. Other than that, the methods of our URL class that you will use are identical to those of the built-in class.

`StringListTester` using `BlueJ`, a window will appear with two text fields at its top. As the labels near the text fields suggest, one is used to create a new `StringList` by simply typing the size of the list into the text field and pressing return. The other is used to add an entry to the list. Again, simply type the entry into the textArea and press return.

The contents of the entire list will be displayed in a text area in the middle of the `StringListTester` window. To test your `getNext` method, press the button at the bottom of the window. The result returned by the method will be displayed in the text field to the left of the button. Create a small list to make sure that your class behaves correctly when the list becomes full.

Step 3: Determining HTTP response types

The final component you need to implement is a class named `HTTPResponse`. Objects of this class will hold the responses your program receives from HTTP servers. The constructor for the class should expect two parameters: a `String` containing the complete response from the web server, and the `URL` used in the request made to the web server. For example, if you have defined a variable toServer that refers to an HTTPConnector and a variable pageToFetch that refers to the `URL` of the next page to process, you could create an HTTPResponse object to examine the page by evaluating

```
new HTTPResponse( toServer.getWebFile( pageToFetch ), pageToFetch )
```

The HTTPResponse class will provide three methods to extract desired information from these responses: `type`, `getEmails`, and `getURLs`. You should start by implementing the simplest method:

```
public String type()
```

> The response your program receives from an HTTP server will begin with a sequence of "header" lines describing the contents of the response. One of these lines will begin with the text "`Content-Type:`". The remainder of this header line will contain a description of the type of data included in the response. This method should return this type information. The complete program will use this method to ensure that it only searches for URL and email addresses in responses that contain HTML.

You should be able to write and debug each of the three methods included in the `HTTPResponse` class independently. When you want to test any of the methods, create an instance of the `HTTPResponseTester` class included in the starter folder. This class will load a sample HTML file included in the starter folder and create an instance of your `HTTPResponse` class using this text. It displays four buttons that can be used to see the contents of the sample HTML or to display the result obtained by invoking your type, `getURLs` or `getEmails` method. Before trying to implement getEmails or getURLs, use the `HTTPResponseTester` class to verify that your `type` method works correctly.

Step 4: Extracting email addresses

You should next implement a method to extract email addresses from the text of an HTML file:

```
public void getEmails( StringList emails )
```

> This method will search the text of a response looking for email addresses. It will add any addresses it finds to the `StringList` passed to it as a parameter.

A simple strategy will enable you to find email addresses. You really don't even need to know anything about HTML to do this. Since all email addresses have @ signs in them, you should start by searching for

an @. Once you find an @, find the first delimiter (blank, quote, new line, etc.) after the @ and the nearest delimiter before the @. Assume everything in between is an email address. Experimentation will lead you to ways to refine your strategy for identifying email addresses. For example, you will probably find that you need to add characters like the less than sign and colon to your list of delimiters. You may also want to check that the string you have extracted is an email address by making sure it contains at least one period. It is fine, however, if your program occasionally extracts something that looks a bit like an email address but isn't really an email address. There is so much variety in the contents of HTML files that it is quite hard to come up with a rule that will reliably extract exactly the email addresses from all such files.

The implementation of both `getEmails` and the method that extracts URLs will involve searching for any of several delimiters including spaces, quotes, less than signs, and new lines. The built-in Java methods `indexOf` and `lastIndexOf` do a good job of searching for a single character, but they were not designed to search for any of several possible delimiters simultaneously. You will, therefore, find it very helpful to define private methods named `indexOfNextDelimiter` and `indexOfPreviousDelimiter` which perform these functions. Since the set of useful delimiters may be different for email addresses and URLs, these methods should each take two strings as parameters. The first string will be the string to search. The second string will be the list of delimiters to look for. The method will use `indexOf` (or `lastIndexOf`) to search for each delimiter separately and then return the index of the closest delimiter found. Like `indexOf`, you may want to support an additional parameter indicating the position where the search should start.

Once you think you have completed the code of the `getEmails` method, you can test it using the `HTTPResponse` class that you used to test your `type` method. Just run the program and press the "Show Emails" button. Even though `HTTPResponse` passes a `StringList` as a parameter when it invokes your getEmails method, the class has been written so that it does not depend on the correctness of your `StringList` class. It includes its own partial definition of StringList. Unlike the `StringList` class we asked you to write, the version used by `HTTPResponseTester` is not based on arrays and therefore would not serve as a good starting point for your own implementation of this class.

Step 5: Extracting URLs

In addition to extracting email addresses from a page, you need to extract the URLs of other pages so that your program can automatically expand its search. For this, you should write the following method:

```
public void getURLs( StringList urls )
```

> This method will search the text of a response looking for links and extracting the URLs found within those links. It will add any links it finds to the `StringList` passed to it as a parameter.

HTML basics

To write the getURLs method, you need to know just a little bit about the structure of HTML. In HTML files, pieces of text that will appear as parts of a web page are surrounded by "tags" that provide instructions on how the text should be displayed. There are tags to center text, tags to make text appear in bold face, tags to make links, and many others.

All HTML tags are composed of text surrounded by a pair of less than and greater than signs. The first letters after the opening "<" describe the main purpose of the tag. For example "<p>" is used to indicate the beginning of a new paragraph. A tag starting with the letter "a" is used to make a link.

Many tags include additional information between the less than and greater than signs. In particular, when making a link, the tag needs to include the URL of the page the link should refer to. For example, to make the text "lecture schedule" refer to the CS 134 lecture schedule, we might include the HTML text

```
<a href="http://www.cs.williams.edu/~cs134/lectures.html">lecture schedule</a>
```

Unfortunately, HTML is case-insensitive and spaces and quotes are optional in many contexts. So we could create the same link by typing

```
<a HREF=http://www.cs.williams.edu/~cs134/lectures.html>lecture schedule</a>
```

or

```
<A Href=http://www.cs.williams.edu/~cs134/lectures.html >lecture schedule</A>
```

The key things to notice are that you should start by applying the `toLowerCase` method to the whole page so that you don't have to worry about case differences, that the URL will be preceded by the text "href=", that the URL may or may not be surrounded by quotes, and that extra blanks can be added after the URL.

Given this brief introduction to HTML, you now know enough to write code to find and extract URLs from a web page. Repeatedly search for the string "href=". Whenever you find such a string, assume that the URL for the link is everything after the equal sign (except for possibly a leading quote) up until either a quote, a blank, a new line or a greater than sign (or any other delimiters you discover as you experiment with your program).

Relative URLs and non-http URLs

There are two additional facts about URLs that your program must address.

Since many web sites are composed of dozens of files all stored in the same directory on the same server, HTML provides a shorthand notation for linking to another file that is located in the same directory. For example, both the home page for our course "`http://www.cs.williams.edu/index.html`" and the lecture page used as an example above are store in the same directory. Using the abbreviated form of URL, we can include a link from the lectures page back to the home page by including the following text

```
<a href="index.html">Home</a>
```

rather than the more complete specification

```
<a href="http://www.cs.williams.edu/~cs134/index.html">Home</a>
```

The short form of URL is called a *relative URL*. The longer form is called a *complete URL*. While your program could translate relative URLs into complete URLs using String methods, there is an easier way provided by the Java URL class.

When we introduced the URL class, we described how to construct a URL using a construction of the form

```
new URL( "http://www.someserver.com/somedirectory/somefile.html" )
```

This form of URL construction can only handle complete URLs. There is a second constructor designed to handle both complete and relative URLs. This alternate constructor accepts a String that it interprets as a URL as its second parameter. As its first parameter, it expects the `URL` of the page from which the second parameter was extracted. If the second parameter describes a relative URL, the constructor completes the relative URL using information from the first parameter. For example, if we had already defined

```
URL pageFetched = new URL("http://www.cs.williams.edu/lectures.html");
```

the construction

```
new URL( pageFetched, "index.html" )
```

would be equivalent to the construction

```
new URL( "http://www.cs.williams.edu/index.html" )
```

If this form of constructor is passed a string that includes a complete URL, the first parameter will be ignored. Thus

```
new URL( pageFetched, "http://www.yahoo.com/index.html" )
```

will be equivalent to

```
new URL( "http://www.yahoo.com/index.html" )
```

When the code of your `addURLs` method extracts a URL from the text of a web page, it should use the `URL` constructor that expects two parameters. We told you that the `URL` that was used to fetch a document from a web server should be included as a parameter when constructing an `HTTPResponse` object. That `URL` should be used as the first parameter of the `URL` constructions performed within your `addURLs` method.

The `URL` class provides one more handy feature. The first part of a URL indicates the type of server the URL refers to. This portion of a URL is called the protocol. You only want to gather URLs that start with "http". The `getProtocol` method associated with a `URL` will return this component of the complete `URL`. You should check to make sure that each URL you extract from a web page uses the "http" protocol before adding it to the list of URLs.

Once you think you have completed the code of the `getURLs` method, you can test it using the `HTTPResponse` class that you used to test your `type` and `getEmails` methods. Just run the program and press the "Show URLs" button.

Step 6: Putting it all together

The stater project also contains a class named `EmailCollector`. It implements the interface for the complete program described in the introduction to this assignment, relying on the `HTTPConnector`, `StringList` and `HTTPResponse` classes. Once you have written and tested these three classes, creating a copy of `EmailCollector` should give you a complete program for harvesting email addresses from the web.

Submission instructions

As usual, make sure you include your name and lab section in a comment in each class definition. Connect to the server "cortand" and log in as guest.

- Select the volume "Courses" to mount and then click "OK." (and then click "OK" again)
- Drag your project's folder into either "Dropoff-Monday" or "Dropoff-Tuesday".

You can submit your work up to 5 p.m. on the day the lab is due. 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. A penalty of 10% per day will be applied for late submissions. If your submission will be more than one day late, you must receive permission to submit your assignment from your lab instructor.