

# HW2 Web

---



*Webviborous SiteSuckeri*

## Part A -- SiteSucker

For part A, you will build a little web client that sucks urls out of web pages.

Due midnight ending Thu May 8th. As before, P/NC students may work in teams of 2, and they are not required to do part B

The WebConnect class represents a connection to a single URL (see the starter file for details). A WebConnect object responds to a doConnect() message that causes it to make an HTTP/1.0 GET request for its URL, and store the results (if we did HTTP/1.1, we would have to deal with chunked responses -- blech). Java includes some built-in classes for HTTP connections, but we are not using them directly. I'd like you to write the HTTP connection code yourself, to get more experience with sockets and to get an under-the-hood sense of how the world's most important protocol works.

The static main() function in WebConnect gives a simple command-line interface to the class. The "-u url" command takes a url, attempts the connection, and prints the resulting "status" line for that url followed by the HTTP response header itself if present...

```
-> java WebConnect -u http://www.stanford.edu/class/cs193i/  
http://www.stanford.edu/class/cs193i/ OK type:text/html time:244  
HTTP/1.1 200 OK  
Date: Wed, 01 May 2002 16:18:09 GMT  
Server: Stronghold/3.0 Apache/1.3.19 RedHat/3014c WebAuth/2.5 (Unix) mod_ssl/2.8.1  
OpenSSL/0.9.6 WebAuth/2.5 mod_fastcgi/2.2.10  
Connection: close  
Content-Type: text/html
```

```

-> java WebConnect -u http://maosucks.stanford.edu/
http://maosucks.stanford.edu/  ERR-CONNECT
-> java WebConnect -u mailto:spam@cs.stanford.edu
mailto:spam@cs.stanford.edu SKIP
-> java WebConnect -u http://www.stanford.edu/class/cs193i
http://www.stanford.edu/class/cs193i ERR-301 type:text/html; charset=iso-8859-1
time:149 location:http://www.stanford.edu/class/cs193i/
HTTP/1.1 301 Moved Permanently
Date: Wed, 01 May 2002 16:34:07 GMT
Server: Stronghold/3.0 Apache/1.3.19 RedHat/3014c WebAuth/2.5 (Unix) mod_ssl/2.8.1
OpenSSL/0.9.6 WebAuth/2.5 mod_fastcgi/2.2.10
Location: http://www.stanford.edu/class/cs193i/
Connection: close
Content-Type: text/html; charset=iso-8859-1

```

The status line is made of several parts, separated by spaces...

**http://www.stanford.edu/class/cs193i/ OK type:text/html time:244**

- First, the URL itself
- A result word summarizing the status code of the GET request. For a successful 200 connection: "OK". If the connection was not opened because the protocol is not HTTP: "SKIP". If the connection could not be made for any other reason (bad hostname, connection refused, ...): "ERR-CONNECT". For any other code, such as 301, append the code number to the string "ERR-", such as "ERR-301".
- If the HTTP header was retrieved successfully, and it included a content-type header, print the type like this "type:text/html".
- If the HTTP header (and possibly body) was retrieved successfully, print the elapsed time to make the connection and retrieve the content, in milliseconds like this "time:244". The method `System.currentTimeMillis()` returns the current time in milliseconds as a long.
- If the result code was in the 300 range, and the header included a location: field, include "location:new-url" at the end of the status.

If the "-a" option is specified instead of "-u", the HTTP body should be printed after the HTTP header, so long as it is a "text/\*" MIME type. The text in the header and body should be converted to use "\n" line endings...

```

-> java WebConnect -a http://www.stanford.edu/class/cs193i/test/basic.html
http://www.stanford.edu/class/cs193i/test/basic.html OK type:text/html time:196
HTTP/1.1 200 OK
Date: Wed, 01 May 2002 16:45:43 GMT
Server: Stronghold/3.0 Apache/1.3.19 RedHat/3014c WebAuth/2.5 (Unix) mod_ssl/2.8.1
OpenSSL/0.9.6 WebAuth/2.5 mod_fastcgi/2.2.10
Connection: close
Content-Type: text/html

```

```
<html>
```

```

<head>
<title>Basic</title>
</head>

<body>
<h1>Basic</h1>

A few basic URLs

<ul>
<li><a href=http://www.theonion.com/>The Onion</a>
<li><A HREF=http://www.stanford.edu/class/cs193i/test/a.html>a absolute</A>
<li><a href      =  b.html>b relative</a> <li><a name=foo href=
"c.html">c relative</a>
<li><a href=mailto:spam-me@cs.stanford.edu>mailto</a>
<li><a href=jeffsad.jpg>Jeff Sad image</a>
<li><a href = "http://www.stanford.edu/class/cs193i" name=bar>missing </a>
<li><a href = http://maosucks.stanford.edu/>no such host</a>
<li><a href = "binky:some/unknown.protocol" >unknown protocol</a>
</ul>

</body>
</html>

```

## doConnect()

The WebConnect object encapsulates a url, and ivars such as "type" and "body" which represent the status items above. All the ivars start out with default values -- the ints are -1 and the strings are all null. When the doConnect() happens, the GET process gets as far as it can, setting the ivars as it goes. We'll give doConnect() a simple linear design -- if it errs out at some point, the ivars that have not been set yet will be left with their default values.

1. Check that the protocol is "http". If not, do not do the connection. Set a flag so that the status string knows to say "SKIP".
2. Attempt to open the socket (start timing)
3. Make the GET (or HEAD) HTTP/1.0 request. Include a "connection:close" in the request, to make it clear that the we do not want a persistent connection.
4. Try to read the HTTP header
5. Parse the result code and type out of the HTTP header. If the process fails before here, the code will still be -1 (which translates to "ERR-CONNECT" in the status string), and the head and body will be null.
6. If we're doing a GET, and the code is 200, and the type starts with "text/", attempt to download the body text.
7. Close the connection and note the elapsed time.

## Milestone

Get -u and -a modes working. Most of the work is in doConnect() to perform the connection, and getStatus() to parse the information out of the HTTP response.

## -x Mode

With the -x "extract" command "-x url", WebConnect should do a connection for the url header and body as above. If the body is present and is text/html, then extract all the <a href=url> urls from the body, do the relative-absolute conversion, and print them one per line...

```
-> java WebConnect -x http://www.stanford.edu/class/cs193i/test/basic.html
http://www.stanford.edu/class/cs193i/test/basic.html OK type:text/html time:723
http://www.theonion.com/
http://www.stanford.edu/class/cs193i/test/b.html
http://www.stanford.edu/class/cs193i/test/c.html
mailto:spam-me@cs.stanford.edu
http://www.stanford.edu/class/cs193i/test/jeffsad.jpg
http://www.stanford.edu/class/cs193i
http://maosucks.stanford.edu/
```

## Parsing

Href urls may look like...

<a href=foo.html> -- simple

<a href="http://foo.html?extra+stuff&hi='hello there'//yo"> -- quotes enable all sorts of chars in the url.

<a name=foo href=bar.html binkymode=true> -- with other bindings in the tag

<a name=foo href=bar.html binkymode=true> -- as above, but with more whitespace

<anthrax mode=true> -- looks like an <a ..>, but it's not

Our practical assumption will be that url hrefs will not include " < >.

Some pages put hrefs in single quotes ('), and allow the url to include double quotes in that case. We will not support that use.

We will not do error checking on the HTML -- your code only needs to work for syntactically correct HTML and hrefs. We need to handle HTML comments <!-- this is comment text <href=blah> -->. We'll just delete them out of the HTML before processing it for hrefs. Make sure your parsing works for the URLs in the pages at http://www.stanford.edu/class/cs193i/test/

## Parsing Strategy

Here's our recommended parsing strategy. The ParseUtils starter file has the basics stubbed out for you. Use Java regular expressions for the parsing.

1. Find and delete the HTML comments `<!-- .....-->`. It would be more efficient to handle the comments and the url extraction in a single pass, but it's much simpler to handle the comments in their own step, so we'll do it that way.
2. Find each `<a` and the `>` that follows it. Extract the whole tag to work on separately.
3. Within the tag, find the `href = url` section. The url may or may not be enclosed in double quotes. If the url is enclosed in double quotes, it may include almost any character, so look for the closing quote (`"`) to mark the end of the url. If the url is not enclosed in quotes, then its end is marked by the first whitespace or `>` character.

## Milestone

Get the parsing working so that `-x` prints out all the extracted hrefs in their raw, relative form first. The URL conversion can be added last, since it's easy.

## URL Conversion

We'll let Java's built-in URL object do the relative-absolute conversion for us -- just use the ctor `URL(base-url, relative-url-string)`. If the base url protocol is not known (such as the `binky:` protocol above), the URL conversion will throw a `MalformedURLException`. In that case, we'll just ignore that URL and not print it in the `-x` listing. You may assume that the initial command-line URL is always an HTTP url, so that its URL ctor will never fail.

## -c Mode

Finally, `-c` "check" mode is similar to `-x` mode, except it constructs a `WebConnect` for each nested URL, attempts a `HEAD` connection for each, and prints its status string. We do not download the content of each URL, we just do a `HEAD` to check that the URL is good...

```
-> java WebConnect -c http://www.stanford.edu/class/cs193i/test/basic.html
http://www.stanford.edu/class/cs193i/test/basic.html OK type:text/html time:221
http://www.theonion.com/ OK type:text/html time:145
http://www.stanford.edu/class/cs193i/test/b.html OK type:text/html time:31
http://www.stanford.edu/class/cs193i/test/c.html OK type:text/html time:23
mailto:spam-me@cs.stanford.edu SKIP
http://www.stanford.edu/class/cs193i/test/jeffsad.jpg OK type:image/jpeg time:19
http://www.stanford.edu/class/cs193i ERR-301 type:text/html; charset=iso-8859-1
time:12 location:http://www.stanford.edu/class/cs193i/
http://maosucks.stanford.edu/ ERR-CONNECT
```

This mode is slower, since it does many more network connections (using `HEAD` vs. `GET` compensates somewhat). You can use `-c` mode to try your solution out

with real web-sites, although you may annoy them with the web traffic, and in some cases their URLs may fall outside of our assumptions. For our testing, we will use pages like the ones at the `cs193i/test` url above.

## Part B -- WebServer

For Part B, you will build a little web server. Part B is smaller than Part A. S/NC students do not need to do part B. The WebServer starter file has some of the basic structure filled in for you.

Our little low-budget web server will not handle all cases, but you can get surprisingly far with just a little code.

### Main

WebServer takes two command line arguments. The first is the port number to use, and the second is the document root to use, which defaults to "." if not specified.

### ServerSocket

Create a ServerSocket for the appropriate port number and block, waiting for incoming connections. We will handle connections one at a time.

### Request

Assume that the client makes a syntactically valid GET request, and parse out the request path. We will just look at the request path, ignoring the rest of the client request. I put my GET-parse method over in the ParseUtil class, so all the regular expression stuff is in one place. Do not worry about error handling -- it will be sufficient to work correctly where the client makes correctly formatted requests.

### 1. index.html

If the request ends with "/", think of the request as having an "index.html" appended to it. So a request for "/" will get "/index.html" and an original request for "/dir/" will get "/dir/index.html". A more sophisticated strategy could look for multiple files in the directory: "index.html", "index.htm" ... but we won't bother with that.

### 2 File

Create a Java File object that represents the document root with the desired path appended: `new File(root + req)`. The File object responds to messages like `exists()` and `isDirectory()` which will be handy for the next steps.

### 3. 404

Check if the file exists. If it does not, return a 404 Not Found HTTP response, and in the body of the page, include the text "Not found: *request*". Some browsers will show this page, and others will just put up a dialog. If the file does exist, continue with the steps below.

#### 4. 301

Check if the file is a directory. If it is, but the original request does not end with a "/", then we need to generate a 301 redirect with a Location: field giving the correct url with a trailing "/". Use `sock.getLocalAddress().getHostAddress()` to get the server's own IP address, to put together the correct url. We use the IP address instead of the local DNS name for the cases where we don't have a functioning DNS name. When this is working correctly, a request for "/dir" will automatically redirect to "/dir/" and so lead the client to pick up the index.html file in that directory when it re-does the request.

#### 5. Simple Cases

Getting past all the ways the request can be wrong, we have simple cases where the request maps to a file that does exist. If the file ends with ".html" or ".htm", send back an HTML response with type `text/html`, and send the whole file as the body. Likewise, if the request ends with ".jpeg" or ".jpg" or ".gif", then send back the appropriate HTML type followed by the whole file. The starter file includes a utility method that sends a local file out a stream.

In all cases, do a `close()` on the socket once all the data is sent and then loop around to handle the next request.

#### Logging

For each request, the server should print a single text line showing the response code number and the original request path:

```
200 /
404 /nosuchfile.html
200 /test.html
301 /dir
200 /dir/
```