

HW 3b LinkTester

Due midnight ending Tue March 5th

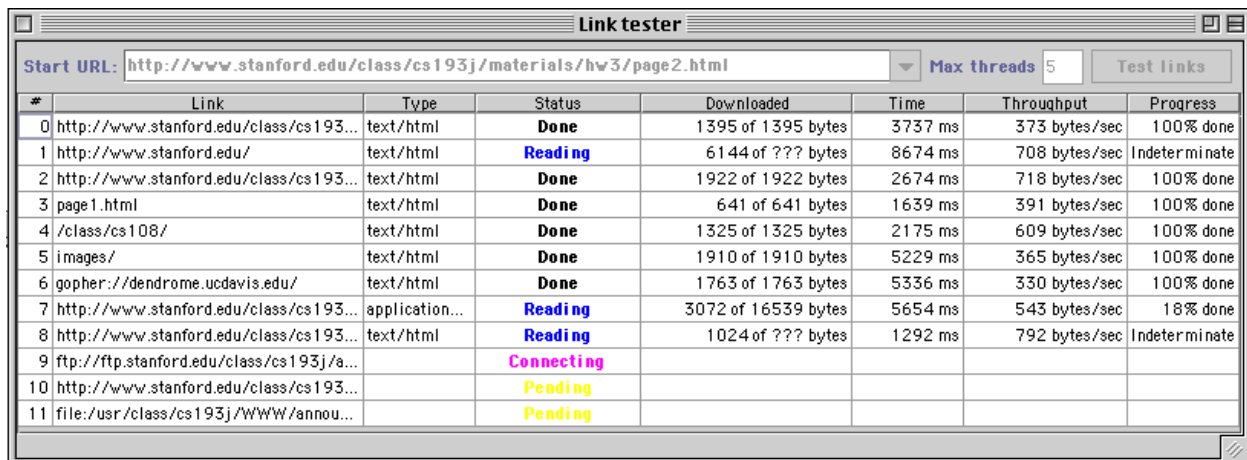
One of the neatest things about Java is its strong support for networking, which makes sense given its positioning as the "language of the Web". This assignment will give you a chance to try out the Java net package to write a simple network program. The program extracts the links from an HTML page, tests each link for validity, downloads the entire contents, and reports the host response time. (Thanks to Julie Zelenski for creating this assignment.)

The LinkTester service

The user supplies a starting URL, and your program downloads that page and scans it for links, each of which is also downloaded. The program tracks information about each link and updates a table with the progress. When finished, the user has a complete list of the all the URLs linked from the start page with information about the validity, size, throughput, etc. of each link.

The program from a user's perspective

The user enters a string that establishes the starting URL. This is the main page from which links are extracted. The user can also give the maximum number of concurrent threads to use for downloading. There is a table listing all the links with the facts for each. This table is updated in real-time as the concurrent downloading threads make progress. Here's a screenshot from our version in action:



The screenshot shows a window titled "Link tester" with a "Start URL:" field containing "http://www.stanford.edu/class/cs193j/materials/hw3/page2.html", a "Max threads" field set to "5", and a "Test links" button. Below is a table with 8 columns: #, Link, Type, Status, Downloaded, Time, Throughput, and Progress. The table contains 12 rows of data, with various status values like "Done", "Reading", "Connecting", and "Pending".

#	Link	Type	Status	Downloaded	Time	Throughput	Progress
0	http://www.stanford.edu/class/cs193...	text/html	Done	1395 of 1395 bytes	3737 ms	373 bytes/sec	100% done
1	http://www.stanford.edu/	text/html	Reading	6144 of ??? bytes	8674 ms	708 bytes/sec	Indeterminate
2	http://www.stanford.edu/class/cs193...	text/html	Done	1922 of 1922 bytes	2674 ms	718 bytes/sec	100% done
3	page1.html	text/html	Done	641 of 641 bytes	1639 ms	391 bytes/sec	100% done
4	/class/cs108/	text/html	Done	1325 of 1325 bytes	2175 ms	609 bytes/sec	100% done
5	images/	text/html	Done	1910 of 1910 bytes	5229 ms	365 bytes/sec	100% done
6	gopher://dendrome.ucdavis.edu/	text/html	Done	1763 of 1763 bytes	5336 ms	330 bytes/sec	100% done
7	http://www.stanford.edu/class/cs193...	application...	Reading	3072 of 16539 bytes	5654 ms	543 bytes/sec	18% done
8	http://www.stanford.edu/class/cs193...	text/html	Reading	1024 of ??? bytes	1292 ms	792 bytes/sec	Indeterminate
9	ftp://ftp.stanford.edu/class/cs193j/a...		Connecting				
10	http://www.stanford.edu/class/cs193...		Pending				
11	file://usr/class/cs193j/WWW/annou...		Pending				

The user enters a URL, sets the number of threads, and clicks the "Test" button, then the link tester is off and running. It first downloads the starting URL. If successful, it scans the downloaded result for any embedded links within that page. Each new link found adds a row to the table and sets up a new download thread (the start URL is the first row in the table). While the download threads read and store the data, the entries in the table are updated in real-time, allowing the user to watch the ongoing progress. When the all downloads finish, the user can enter a different URL and do it all over again.

Note that while a link test is in progress, the controls (URL, max threads, test button) are all deliberately disabled and only re-enabled after the current test completes (use `component.setEnabled(boolean)`). During the download, the rest of the user interface, the scroller on the JTable, for example, remains responsive and snappy since the downloading work is going on via background threads instead of tying up the swing thread.

Each table row corresponds to a link from the starting URL. For each, the table should display:

Content type: The MIME type as returned by `URLConnection` (or null if it can't provide it)

Status: A string that indicates the current state of activity for this URL. You can decide exactly what status strings you would like to use, but we expect you to indicate various facts such as "Pending" when this URL is not being actively downloaded because too many other threads are running, "Connecting" when trying to establish the connection, "Reading" while downloading, and "Successful" for when fully downloaded with no errors. For non-successful URLs, a status such as "Malformed", "Failed to connect", "I/O error in downloading" should provide the reason for failure. The status info is updated as the URL passes through the various stages.

Bytes downloaded: This shows the number of bytes downloaded so far. For those URLs that provide a known content-length you should also state the total number of bytes expected to be read. This count is regularly updated after each new chunk of data is read.

Time downloading: The number of milliseconds spend thus far in connecting and downloading this document. This number is updated along with the byte count during downloading.

Throughput: The count of bytes downloaded divided by the time used. This is updated at the same time as bytes and time.

Progress: A number between 0 and 100% that indicates the percent of the document that is downloaded. For URLs that don't provide a known content-length (i.e. we don't know in advance how many total bytes will be downloaded), this entry can be "Indeterminate" or some other such designation.

The program from an implementor's perspective

The above description may sound a little daunting, but the useful built-in objects are going to give you a good leg-up. For example, one task that sounds like a chore (and usually is with most tools) is parsing a string representation of a URL, setting up a connection to access the resource, and reading its contents. However, with just a few lines of Java code, you can convert a `String` into a `URL`, open a connection on it, get a stream on its contents and read the data just like you would from any other input stream. A lot of work is going on behind the scenes, but pretty much the only networking classes you need for the `LinkTester` are `URL` and `URLConnection`. You'll find both of these a delight to use because they handle some fairly messy and complicated tasks but only require you to learn a very clean and simple interface. Bravo to their designers!

URL and URLConnection

A Uniform Resource Locator (URL) is a scheme for encoding a network resource. It identifies what host to find the resource, the path on that host and the network protocol to retrieve it. The basic format of a URL is `protocol://sitename/path`, here are a few sample URLs:

```
http://www.ibm.com/index.html
ftp://ftp.stanford.edu/class/cs193j/assignments/hw3/
file:/usr/class/cs193j/other/grader_assignments
mailto:cs193j@cs.stanford.edu
```

Java's `URL` class nicely handles dealing with the needs for each of the different protocols. Given a `String` for the URL, it sorts out what host to contact and what communication protocol to use.

One detail of which you need to be vaguely aware is the handling for partial or relative URLs. The sample URLs listed above are all "full" URLs that completely specify the host and path to the resource. Often, the links within a page are relative links, not absolute. For example, if you

encounter a link to "list.html" (not beginning with a '/'), this relative URL picks up the protocol, host, and path of its referring document. So the relative URL "list.html" in the document "http://www.ibm.com/tools/index.html" becomes "http://www.ibm.com/tools/list.html" in its full form. Root-relative URLs like "/images/red.gif" (beginning with a '/') pick up the protocol and host of their referring document, but not the rest of the path. So a link to "/images/red.gif" in the page "http://www.ibm.com/tools/index.html" becomes "http://www.ibm.com/images/red.gif".

However, we're boring you with this mostly for your own information. You don't need to be clever about figuring out which links are relative or root-relative. If you research the URL constructors, you'll find one that specifically takes a string and a "context" URL. It is designed for just this case—you have a link that came from a referring document (the context URL). If it is a partial link, the constructor will take the missing pieces from the context URL and otherwise, it uses the absolute URL. Either way, it produces the combined URL for you. Pretty nifty!

Once you have the URL, setting up the URLConnection is just a few lines of code. Each time you encounter a new link from the starting page, you will set up another downloading thread to set up a connection and download this new link. Be sure to keep track of the URLs that have been downloaded so that you don't incorrectly and wastefully download a URL more than once. To do duplicate detection, you can use the URL equals() method. It doesn't catch all cases (it can be fooled when the capitalization is different, for example), but it is good enough for our purposes.

Content type and length

Documents on the web are identified by content-type using MIME classifications such as text/html, image/gif, and application/pdf. This is not determined by the path name, but by asking the URLConnection object to identify the type. Some servers and protocols may not reliably respond when asked for the content type and just return null. Don't make a special case for this, just indicate null in the content type field.

The content length method returns the total numbers of bytes in the document. The connection returns -1 if the server doesn't provide length information or can't determine the total length in advance. **Known bug:** Asking the URLConnection for the content length before opening the connection may return an erroneous -1 result, so you should order your calls to avoid this.

Reading and keeping time

Once you have the URLConnection set up, getting an input stream on that connection is a trivial task. Downloading is a matter of reading from that stream until you reach the end. You should use the version of the InputStream/Reader read() method that allows you to read a chunk of data at a time. Set your download thread up to read in chunks of 500 bytes until it reads the end. Amass the results by appending into a StringBuffer and you will have re-constructed the whole document! After each chunk is read, it will be your chance to update the bytes, time, throughput, and progress for this link in the table. To track how much time has elapsed, use the currentTimeMillis() static method of the System class. By saving the current time before the thread starts to read, and then subtracting from the current time after the read finishes, you can roughly determine how much time it took to read that chunk and use that information to update the table. Although using the system clock like this is not all that precise, the approximation is good enough for our purposes

The java.io package contains rich and extensive support for all sorts of I/O needs, but it can be unexpectedly difficult to accomplish simple things because you have to wade through many different classes. We have tried to make the I/O portions of the assignments fairly straightforward so they don't cause you much grief.

Problematic links

All too often on the web, you run into invalid links. These can be links that are ill-formed (perhaps due to typos on the part of their author) or refer to pages that have been renamed or moved or hosts

that are down. Sometimes, what appears to be a valid link can even go sour in the middle of reading due to a poor connection and throw an I/O exception. Your program should make one attempt to access each link and if any problem gets in the way, the downloading attempt for that link is abandoned. The status for this URL should reflect an error occurred in downloading. You can either leave the information about the partial download (num bytes, time, throughput) or blank it out, whatever you prefer.

If you can connect to the URL and the connection object returned is of type `URLConnection` (it is acceptable to use an instance of `URLConnection` here), ask it for the http response code before downloading the document. If the code is the `HTTP_OK` code (200) go ahead and get the stream and download the contents normally. All other codes (404 Not found, 403 Permissions, etc.) indicate you should not attempt to download the document and should quit working on this link, reporting the error code in the status field.

HTML LinkScanner

From your past experience and working on hw1, you are all reasonably familiar with the basics of HTML. An HTML file contains the document text along with embedded format codes for text styles, images, tables, links to other URLs, etc. HTML formatting tags are the words enclosed in angle brackets that are intermixed with the text, such as `<BOLD>` or `</TABLE>`.

Rather than have you mess further with the nasty bits of HTML for this assignment, you will instead just use our provided `LinkScanner` class to search for URLs within an HTML document. The `LinkScanner` knows about the structure of HTML and can extract the URLs from those tags that contain links, such as anchors, image maps, and frames. For example, here is an anchor tag:

```
Go to <A HREF="http://www.pizzahut.com">Pizza Hut</A>!
```

When a new scanner is constructed, the client provides the input stream or reader to scan (which came from an opened file, a URL connection, a stream created on a string, etc.) The client repeatedly messages the `LinkScanner` for the `nextLink()` to retrieve URL strings one by one. It returns null when there are no more URLs found.

The thread pool

As noted in the earlier description of the program, downloading goes on "in the background." The user should be able to scroll the table and click around while the pages are being read. To accomplish this concurrency, for each page to be downloading (including the starting one), you should start up a separate thread to handle opening a connection to the site, retrieving the contents, and updating the info in the table. However, some web pages may contain many, many links, and it is neither polite nor wise to create an exorbitant number of threads to swamp the network.

Thus there is a limit on the maximum number of downloading threads. For example, if the user enters 5 into the max threads field, then at most 5 concurrent threads will be downloading at any given moment, other requests will queue up behind them. This provides a reasonable way to limit the stress and avoid overtaxing any machines.

To support this, you will create the general class `ThreadPool` to control the thread activity. A thread pool is created with a given maximum number. When a client needs a new thread, it passes the request (in the form of a `Runnable` object) to the `ThreadPool`. If there are fewer than the maximum number of active threads, the `ThreadPool` immediately starts up a new thread to handle the `Runnable` object. If the maximum number of threads are already running, the `ThreadPool` will add the `Runnable` object to a queue of pending requests (using a vector or list here would be fine). When another thread finishes, the `ThreadPool` will take one of the pending requests and start it in a new thread. Note that the `ThreadPool` class should be a generic thread manager, not specific to

downloading or URLs, it only knows what each thread should do by virtue of the Runnable object passed in by the client.

The ThreadPool sounds simple enough, but there are some tricky issues here that you need to think through. If a client requests a new thread and it cannot be immediately started, the client should not be blocked (i.e. the request made to the ThreadPool should return immediately either way). There is no generic notification posted when a thread exits, so you will have to be a bit clever about recognizing when one of the threads in the pool has finished so you can start another. Also, you need to be quite careful about synchronization of any shared resources here since multiple threads will most certainly access the ThreadPool object.

The ThreadPool will also need to provide one last feature which is tracking when all threads in the pool have completed so that the controls can be re-enabled after the current test finishes. Likely the first thread (the one that downloaded the start URL) will work with the ThreadPool to detect when all threads have finished so it knows when to re-enable the controls.

Synchronization

The downloading threads in the program operate pretty much independently of each other, but they may at times share access to some of the same data, which means you will need to take some precautions. You should take a careful look at the objects you are using (Collections, URLs, etc.) to learn what precautions they already take and whether you need to do anything extra when using them. (see on-line docs for specs on which methods are synchronized). You may also need to synchronize some activities in your LinkTester class as well to avoid any critical sequences of calls being interrupted or interleaved in inappropriate ways. The wait and notify features of synchronization will likely come into play in coordinating the actions of the ThreadPool.

LinkTester user interface

Your LinkTester need not look exactly like ours. It needs to allow the user to specify the starting URL and the maximum threads, and include a table to show the progress, but the details of the layout, arrangements, fonts, colors, controls, etc., are up to you.

The general idea is that you will build your own TableModel implementation that maintains a collection of rows, where each row object manages the data for one link. The thread associated with that link will be messaging that row object to update the values as progress is made. In turn, the row object will fire change notifications that cause the table to update. The design strategies handout contains some general hints and suggestions about building a TableModel. Given the restrictions about accessing Swing components from multiple threads, you will need to funnel actions on the table through the Swing invokeLater method so that you don't cause the Table to collapse in a heap of trouble from being accessed simultaneously by multiple threads and the swing thread.

Your table doesn't need to be particularly fancy. In general, we aren't expecting you to fuss a bunch trying to make the table look pretty. The contents of all table cells can be left-aligned, drawn in black, and in the default font (i.e. using all the default attributes as supplied by the table). The columns can just be the default width— the table will evenly divide the space among the columns, this may look a little silly, but you don't need to change this to please us. We don't expect much beyond a really simple default table. It would be good to set the column names and be scrollable, but that's about all for required frills.

However, if that doesn't satisfy you and you're feeling ambitious and want to learn more about Swing, you can certainly read up and do more! For example, setting the preferred width on each column would allow you to apportion the horizontal space more sensibly. Check out how you can set the column class or change the cell renderer to control how various cells are displayed. For example, changing the color or font when rendering the status for different states makes a nice visual indication when a link transitions between states. Even fancier, you could install a

JProgressBar as the rendering component for the progress column and visually animate the download progress for each link.

Optional HTML viewer

For the most part, this is a threads and networking assignment, so this last part is entirely optional and totally irrelevant, but neat nonetheless. When setting up your window, you can add an HTML pane at the bottom of the frae with something like this:

```
editor = new JEditorPane("text/html", "");
editor.setEditable(false);
scrollpane = new JScrollPane(editor);
scrollpane.setPreferredSize(new Dimension(400,300));
box.add(scrollpane);
```

Then if you respond to the row selection notification `valueChanged()` message, you can grab the downloaded contents for that row and put it in the HTML editor.....

```
ListSelectionModel lm = table.getSelectionModel();
lm.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
        int rowIndex =
            ((ListSelectionModel)e.getSource()).getAnchorSelectionIndex();
        if (rowIndex >= 0)
            // get contents downloaded for link at rowIndex
            editor.setText(contents);
    }
});
```

The HTML editor is a work in progress. It can successfully render some simple pages, but eventually blows itself up with exceptions and running out of memory. It's one of the most active areas of Swing development, so your same code should run a lot better in the future. You can leave the HTML feature in — we won't mess with it.

Requirements summary

Like we did for the earlier homeworks, a summary list that might help you manage the details:

General

- The project you submit should include all necessary files for both programs in one directory. We should be able to issue the command `javac *.java` in your project directory and all files should compile cleanly.
- Your files should be readable on UNIX (e.g. transferred correctly, proper end-of-line chars).
- You should have a `LinkTester` main class that we can run with `java LinkTester`.

Link Tester user interface

- The default value for the maximum number of threads is 5. You can assume the user will only enter valid numbers in the threads field (i.e. you do not have to do any bullet-proofing of the field).
- The controls for configuring a link test (choosing the URL, setting the max threads, the start button) should be disabled while a test is still in progress or paused. These controls should be enabled only in-between tests.
- You do **not** have to provide any form of stop feature. Once a test is started, you may assume it must to run entirely to completion before another test can be configured and started.

- The LinkTester window should be resizable. The layout and components should gracefully scale as the window is resized. We will not maliciously resize the window so small that it becomes ugly .
- Clicking on the close box in the window title bar immediately quits the program.
- The table should include columns for the link number (just assigned sequentially), the link text, content type, status, size, time, throughput and progress. The columns should be appropriately labelled. The table should be scrollable.
- The rows of the table are updated in real-time as the background threads progress. When a link has not yet started, the entries for time, bytes, etc. can be "0 bytes read" or blank, whichever you prefer. Similarly, if downloading fails midway, you can leave the information at its last update (e.g. "100 of 1000 bytes read") or blank it out.

Accepting and rejecting links

- If the starting URL given by the user is invalid, you do not need to do anything special. Just give up on this URL the way you would any invalid link, and thus the test will immediately finish and there will be zero URLs successfully downloaded and one failed URL.
- Each unique link should have its own row in the table and be downloaded at most once. There should not be any duplicate rows. (i.e. each unique link appears once and only once).
- There are many reasons a URL may fail to download:
 - It is ill-formed, i.e. the URL constructor throws a MalformedURLException exception on the link string. If you aren't able to construct a URL object, you do include a row in the table for it, but you will not be able to download anything and the status for this link should be "Malformed" or some such indication.
 - You fail to establish a connection to this URL at all. Again, it still has a row in the table, but you won't download it and the status will be something like "Connect failed".
 - The connection made was of HttpURLConnection type and the response code was not the 200 success code. You should not download the contents and should set the status to indicate the response code failure.
 - An error is raising when reading from the URL connection's stream and interrupts the downloading. You can either clear the in-progress information about this link's throughput and size or leave it half-finished. Either is fine with us. The status should indicate some sort of I/O failure.
- Do not make special-cases for any particular URLs (mailto:, gopher: etc.). If you can create the URL and connect to it, then read and download what you find there.

The thread pool

- The thread pool has a constructor that establishes the max threads for this pool. It should be possible to create multiple instances of ThreadPool which do not interfere with one another.
- Within each pool, there should never be more than its maximum number of threads running.
- Any Runnable object can be passed to the pool when making a request for a new thread. The new thread is created and started if maximum hasn't been reached, or the runnable object is queued and will be started later when other threads complete.
- The ThreadPool should provide a method that blocks until all threads executing in the pool finish running and there are no more waiting requests in queue. (This method can be used to determine when to re-enable the user interface.)