UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **M. Brudno**
Summer 2000

Project #2*

Due: Monday, July 24, 2000

Please read this handout *before* you start coding. It will answer many questions you may have along the way.

This project involves developing a site searcher. One prominent feature of 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 put Java's networking skills to good use in writing something impressive. For this assignment, you will write a Java program that can provide a site-restricted search on demand for any given site. Given a starting URL, it will index words from that page and follow any links from that page (as long as they still point somewhere on the same site) and index those pages as well, to produce an index for the entire site, page by page.

The first part of the project entails developing a *Multiset* data structure for storing an ordered collection of elements. The second part of the assignment will use that Multiset for organizing a site-restricted search on demand for any given site. The first part of the assignment gives you a chance to put your newfound knowledge of interfaces, inner classes, and exceptions to work. You should be able to start on this one right away and make a lot of progress. The second part involves threads and networking, which we should briefly discuss in class soon.

1 From the User's Perspective

The program begins by asking the user to enter a string which names the starting URL. This establishes the main page from which the search is launched. That page is "indexed" and any links from that page to other pages are also indexed. "Indexing" means breaking up the text of the page into its component words and storing references which tracks which words appear in which pages. The user is asked a few questions about how to limit the search. It is assumed that the search be host-restricted, so that only links to pages on the same named host will be included. The user can further constrain the search to be path-restricted by specifying a prefix. If a prefix is given, only files on that host that start with the given prefix will be included in the index. Lastly, the user is asked to limit the maximum number of URLs to index from the site. This provides a stop-gap measure to give user means to prevent any out-of-control indexing on large sites. For example, if the user answers 10, then only the first 10 distinct URLs found will be indexed and that's it. With the search criteria established, the program starts building the index, but rather than waiting until all the indexing is done, the program immediately goes into

*This project was written by Julie Zelenski for CS 193J at Stanford

an interactive loop that allows the user to enter words to be looked up in the index so far, while the indexing continues in the background. When the user enters a word, the program reports the number of matches and then ask the user if they would like to see the list of matches. The full list is the URLs, one per line, listed with the count of occurrences of that word in that page, sorted from highest occurrence count to lowest.

For example, here is an excerpt that shows what we expect:

```
> java SiteSearcher

Welcome to the Java Site Searcher service!
What is the starting URL? http://www-inst.eecs/~cs61b/index.html
What is the restricted prefix (hit RETURN for no prefix)?
What is the maximum number of URLs to index? 25

Status: 0 URLs fully indexed so far, 1 URL in-progress
Enter a word to lookup ("quit" to quit): geek
Found no matches found for "geek".

Status: 2 URLs fully indexed so far, 2 URLs in progress
Enter a word ("quit" to quit): the

Found 2 matches for "the". See them? y
http://www-inst.eecs/~cs61b/lectures/review.html (90 occurrences)
http://www-inst.eecs/~cs61b/index.html (15 occurrences)
http://www-inst.eecs/~cs61b/diatribe.html (13 occurrences)

Status:3 URLs fully indexed so far, 2 URLs in progress
Enter a word ("quit" to quit): brudno
Found 1 matches for "brudno".
See them? (y/n): n
```

Because of the non-determiistic nature of Java threads, this project cannot (and hence will not) be tested automatically (except for the Multiset). However, please have your program mimic our's behavior closely in order to make it easier for the readers.

Notice it gives an update on the background indexing activity each time before prompting the user to enter another word. The status line reports the number of URLs that have been finished (fully indexed), as well as a count of those currently being indexed. This status is a just a snapshot at that particular instant, those numbers are continually changing as the index threads do their work. The in-progress count may at times include URLs that will eventually "fizzle" out because they cannot be opened or aren't text documents. When all indexing is finished, in-progress should be 0 and the indexed count should be the number of URLs that were successfully accessed and indexed.

In the example shown above, note that it is possible to have a certain number of matches when the word is first looked up, but have more references by the time the list is printed out. Because the indexing continues concurrently during the searches, this sort of behavior is expected.

2 Multiset

Arrays are great for storing a set number of elements. However, one often needs an array to grow as more data is gathered. The Vector class in the java.util package implements a growable

array of objects. Like an array, it contains components that can be accessed using an integer index. Additionally, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

As handy as Java's built-in Vector is, it's not perfect for every collection need. For example, consider an unordered collection which contains many duplicated elements. If using a Vector, you'll be storing a reference each time an element appears, scattered throughout the vector. If you wanted to know how many times an element appears in the collection, you need to iterate and count the matches, which could be inconvenient and inefficient. It is also awkward to do such tasks as determine the number of distinct elements or remove all occurrences of an element. If you didn't care about the ordering of your elements within the collection and only need to know the unique elements and the number of times each appears, a more compact and easily managed representation would be the "multiset" which keeps only one reference to each element and tracks the count along with that element.

Your first job for this project is to implement a multiset object using a Vector as your underlying data storage and an inner class to bundle each element with its count. In the Java tradition of building generic modular components, the multiset should accommodate elements of any object type. It should place no restrictions on the number of distinct elements that can be held, but can limit the number of occurrences to the maximum value that can be represented in an integer. It should assume that the `equals()` method for an object can be used to determine if two elements are the same.

2.1 Operations

The operations your multiset class must support include:

2.1.1 Basic Methods

- `Multiset()`
  A simple zero-argument constructor that creates a new empty multiset.

- `int numElements()`
  Return the total number of elements in the multiset. This count includes duplicates, e.g. a multiset containing {2, 12, 2, 2} has 4 total elements.

- `int numDistinctElements()`
  Return the number of distinct elements in the multiset. Duplicates are only counted once, e.g. a multiset of {2, 12, 2, 2} has just 2 distinct elements.

- `int countOccurrences(Object o)`
  Returns the count of the number of times that the given object appears in the multiset. Will be zero if no such object is found at all.

- `void add(Object o)`
  `void add(Object o, int numOccurrences)`
  Adds a new element into the multiset., either just one occurrence or multiple occurrences if using the two-argument version. If the element doesn't already appear in the multiset, it is entered with the appropriate occurrence count. If the element already appears, its frequency is incremented by the newly added quantity. If the count given to the two-argument version is 0 or negative, the method should throw an IllegalArgumentException. Like all of the standard Java collections, a reference to the object is stored, no deep copy or clone is made when a new element is added.

- `void remove(Object o)`
  `void remove(Object o, int numOccurrences)`
  Removes an element from the multiset, either just one occurrence or multiple occurrences if using the two-argument version. If the element doesn't appear at all in the multiset, no changes are made and no error results. If removing will decrement the occurrence count to zero (or below), the element is completely removed from the multiset. No error is raised if the count is larger than the actual number of occurrences. If the count given to the two-argument version is 0 or negative, the method should throw an IllegalArgumentException.

- `void removeAll(Object o)`
  Removes an element and all its occurrences from the multiset. If the element is not found, no changes are made.

- `Enumeration elements()`
  Returns an object which allows the client to enumerate through all of the elements (including duplicates) of the multiset using the standard Enumeration interface. The enumeration is free to visit the elements in any order which is convenient, but each element is visited exactly once during an enumeration cycle. For example, enumerating through the multiset {2, 12, 2, 2} would cycle through all 4 elements in some unspecified order. Like all standard Java collections, changing the collection (adding and removing elements) during an enumeration cycle can produce unpredictable results.

- `Enumeration distinctElements()`
  Returns an object which allows the client to enumerate through all of the distinct elements of the multiset using the standard Enumeration interface. The enumeration is free to visit the elements in any order which is convenient, but each unique element is visited exactly once during an enumeration cycle. For example, enumerating through the multiset {2, 12, 2, 2} would cycle through the 2 distinct elements in some unspecified order.

2.1.2 Callback interface methods

In addition, your Multiset must support a few operations that use client callbacks via interfaces to manipulate the elements. One interface is a comparison routine that given an element and the count of occurrences and another element and its occurrences will return the relative ordering of the two elements, either a negative number if the first should come before the second, zero to indicate they are equal, and a positive number if the first should come after the second. This is similar to using functions as objects in Scheme, or, for you C programmers, it is also similar to the way we would typedef a function pointer callback in C for a sorting routine.

```
public interface Compare {
  public int compare(Object o1, int numOccurrences1,
                     Object o2, int numOccurences2);
}
```

The second interface is for a mapping routine that will be passed an element and its count of occurrences in the process of iterating over the multiset.

```
public interface Mapping {
  public void map(Object o, int numOccurrences);
}
```

Both of these interfaces should be nested definitions inside the Multiset class so that the full name would be Multiset.Compare, to strongly associated these interfaces with the class and to

clearly distinguish them from uses of the names Mapping or Compare anywhere else. The map method will use the above two interfaces. Mapping is an alternate way of performing enumeration over the multiset. In the usual iteration pattern, the client asks for an Enumeration and then manually iterates through calling nextElement and hasMoreElements in a loop to perform some operation on each member in the collection. Instead, the looping could be done by the implementor (the person writing the multiset class) and as it visits each element, it calls the client-supplied callback method to do whatever particular action needs to be done to the element. This type of mapping support can be used to print out all the elements, count all the elements that appear an even number of times, etc., any type of operation that can be expressed as an action done to each element in turn. (This is a Java equivalent for the Scheme mapcar).

- `void map(Mapping mp)`
  Given an object which implements the Mapping interface, this method will iterate over the multiset and call its map method once for each distinct element passing the element and its number of occurrences. For example, in the multiset "Apple", "Banana", "Apple", "Apple" the map method would be called once on "Apple" with count 3 and again with "Banana" and the count 1. In this version of the map method, the distinct elements are not visited in any specified order, the implementation is free to do what is most convenient.

- `void mapInOrder(Mapping mp, Compare cmp)`
  This method operates similarly to the above map method with regards to use of the mapping callback, but takes an additional parameter which can compare elements in the set to determine in which order to visit them. The Compare object is given two elements and their count of occurrences and returns an integer (negative, 0, or positive) that establishes the ordering between the two. The mapping operation will first visit the element which was determined to be "before" all the other elements, and then onto the next element and through all the distinct elements to the one that was determined to be "after" all the others. Elements that the callback reports to be equal (0 result) can be visited in any order.

As a convenience for the client, you will provide a few useful callbacks already implemented for the standard variants one might want to perform with the mapping operations: printing each element and mapping in order from low to high frequency (or the reverse). These should be provided as inner classes, defined within and strongly associated with the Multiset class, but made public so as to be available for use by clients.

- `PrintElementAndOccurrences`
  This class implements the Mapping interface and prints the element along with its occurrence count on a line by itself. For example, if the elements were strings, it would print "`Apple (3 occurrences)`". Using this object in conjunction with the map method will make it easy for the client to neatly print the contents of the multiset with just the call:

  ```
  myset.map(new Multiset.PrintElementAndOccurrences());
  ```

- `SortByIncreasingFrequency` and `SortByDecreasingFrequency`
  These classes each implement the Compare interface and determine the order of the elements by considering their relative frequencies. When sorting by increasing frequency, elements with lower occurrence counts are considered to be before those that are higher and reversed for decreasing frequency. Elements that have the same frequency are reported as equal (and thus, can be mapped in any order). Using this object with the mapInOrder method would produce a printout of the elements in order of increasing frequency with this call:

  ```
  myset.mapInOrder(new Multiset.PrintElementAndOccurrences(),
                   new Multiset.SortByIncreasingFrequency());
  ```

2.2 Testing

You definitely want to thoroughly test your Multiset in isolation before integrating it into the later part of the assignment. Inventing comprehensive test cases and exercising your code to find the lurking problem is an important part of the development process. Try storing Integers or Strings and try all combinations of activities in various rearrangements. Define new callbacks and make sure everything works smoothly, no matter what the base element type. Definitely do not skip this step and jump right into using the multiset in the rest of the homework. It is practically impossible to be developing and debugging a larger program when you're not even sure if the underlying utility objects can be trusted. As discussed in one of our first sections, a handy place to put your testing code is in a main method declared on the class itself, like this:

```
public static void main(String[] args) {
  Multiset set = new Multiset();
  set.add("Apple");
  set.add("Apple");
  set.add("Banana");
  set.mapInOrder(new Multiset.PrintElementAndOccurrences(),
                 new Multiset.SortByIncreasingFrequency());
}
```

This will make it convenient to run just this test code in the interpreter and allows you to keep the test code with the class being exercised. You can leave in testing methods in your submission as long as they don't excessively clutter the rest of your code.

3 Site Searcher

In Section 1, I showed how your site searcher might work. The task may seem daunting at first, but Java does much of the ugly work for you. You will use the standard URL object to translate a string into a network resource, the URLConnection to open a stream on a URL, a StreamTokenizer to separate a stream into words, and so on.

For example, one task that sounds like a chore (and usually is in most languages) would be figuring out how to take a string representation of a URL and managing to set up some connection to access the resource and read its contents. However, in Java, with just a few lines of code, you can convert a String into a URL, open a connection on it, get a stream on its contents and then read the data just like you would from any other input stream. Java does a good job of supporting these common networking operations in pretty straightforward ways.

Another obstacle concerns breaking up the pages into words. We'll provide some help for this in the form of an input stream scanner class (more information in the I/O section below) which will make it simple to pull the words out one by one. For each word, you will keep a Multiset which tracks the pages that contain that word. Duplicate references are counted, so if the word "the" occurs 10 times in the CS61B home page, it will have 10 occurrences for that page tracked in the multiset.

In order to store and lookup by word, you could use a vector and use the will use a Vector and search for the strings with the `indexOf(Object elem)` method. A better (more efficient) method is to use a data structure called a *hashtable* which maps *keys* to *values*. The word is used as the key and the value is a multiset of URLs which reference that word. You should only bother to track words that are at least 3 characters long in your index and will not index any words that appear inside the HTML tags. The index should not be case-sensitive (e.g. BrUdNo and brudno are considered the same) when storing or looking up words for the user. The easiest way to do

this is to convert all words to a standardized form, say all lower-case, before storing or looking them up.

Java has its own implementation in `java.util.Hashtable` and we have provided a couple routines which call the specific Hashtable routines for you in `SiteSearcher.java`. `addToTable(String word, URL newURL)` records the reference of word in a URL in our table, adding the reference to the corresponding set. `getUrlsReferenced(String word)` retrieves the Multiset with all of the URLs that have been added to our table for a particular word.

Each time you encounter a new link while indexing a page, the link is checked to see whether it meets the necessary criteria for inclusion, and if so, another indexing thread is started up to index this new link. Be sure to keep track of the URLs that have been indexed (a Vector will do just fine) so that you don't incorrectly and wastefully count a page's references more than once.

The interactive loop which allows the user to look up words in the index should be quick work for you with the formidable powers of the Hashtable and Multiset at hand.

3.1 HTML

Documents that are published on the Web are formatted in HyperText Markup Language (HTML). An HTML file contains the text to be displayed along with embedded format codes for bold, italic, images, tables, links to other URLs, etc. HTML is a simple language, but can be rather crufty. You don't really need to know much HTML for this assignment, but if you're interested, check out the beginner's guide to HTML at `http://www.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimerAll.html/`. HTML formatting tags are the words enclosed in angle brackets that are intermixed with the text, such as `<HTML>` or `</TABLE>`. Many tags come in pairs, where an opening tag marks the start of a format sequence and a closing tag marks the end, such as `<TITLE>` and `</TITLE>` pairs. The text in-between those two tags will be marked as the document's title. When tags come in pairs, the closing tag has the same name as the opening one except that it is prefixed by a slash. Our provided Scanner class (see below in I/O section) knows to extract tags from a stream (a tag will be everything inside a pair of angle brackets). Most tags will be discarded when indexing and not indexed. The only tags we are actually interested in are the "anchor" tags which are used to mark a piece of text as a link to another page. An anchor tag might look like this:

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

You need to extract the URL referenced in an anchor tag so you can consider that page for indexing as well. Figuring out which tags are anchors and how to pick out the URL from inside such a tag is a somewhat grungy task, so we just gave you a `findURLinTag()` method as part of our Scanner class that returns a URL if one is found inside a given tag or null otherwise. Don't forget to just call upon it when needed!

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

```
ftp://ftp.cs.berkeley.edu/pub/teaching/README
file:/home/ff/cs61b/handouts/proj2.pdf
mailto:cs61b@cory.eecs.berkeley.edu
gopher://portfolio.stanford.edu/StanfordLife/
http://www.whitehouse.gov/White_House/Tours/Welcome.html
```

Although you might expect you would have to figure out what to do for each of the different protocols, you'll be relieved to learn that Java's URL class nicely just handles the de-

tails for you. Given a String identifying the URL, it handles sorting out what host to contact and what communication protocol to use as needed. The only detail of which you need to be somewhat aware is a little bit of extra handling needed for partial or relative URLs. The above sample URLs are all "full" URLs which completely specify the host and path to the resource. More often with a complete site, the links within are relative links, not absolute. For example, if you encounter a link to "products.html" (not beginning with a '/'), this relative URL picks up the protocol, host, and path of its referring document. So the relative URL "products.html" in the document "http://www.ibm.com/hardware/index.html" becomes "http://www.ibm.com/hardware/products.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 document "http://www.ibm.com/hardware/index.html" becomes "http://www.ibm.com/images/red.gif".

However, we're boring you with this mostly just for your own information. If you look at the constructors provided in the java.net.URL class, you'll find there is one that specifically takes a relative link and its base URL document and combines them together into the absolute URL for you. Pretty swift! If you're hungry from more details on URLs, check out `http://www.ncsa.uiuc.edu/demoweb/url-primer.html` and `http://www.w3.org/Addressing/Addressing.html`.

3.2 Network Operations

The only two networking classes you need to be concerned with for this assignment are java.net.URL and java.net.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!

The URL class takes a String representation and parses it into the proper host, protocol and file components. You construct a URL from either an absolute reference or a reference relative to some other URL. URLs can be compared using `equals()` and have getters to retrieve the host, file, protocol, etc. When restricting a search to pages on a particular host, you can check that any newly found URL has the same host as the original starting page, and when restricting to the same path, you can check that a URL's file path begins with the required prefix. To do duplicate detection, just 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.

A URL object just describes a place and means to get to an information source, to actually connect up and read the data, you use the URLConnection object obtained from the openConnection method of the URL. From there, you can learn the content-type of the URL, the modified date, the size, etc. of the file and can get an input stream to read its contents.

Documents are identified by content-type using MIME classifications such as text/html, image/gif or application/pdf. This is not determined by the path name, but by asking the URL connection to identify the type. Since our indexing is only able to parse text documents, you should not attempt to index any documents that don't have a content type beginning with the string "text", just ignore links to any such non-text documents when building the index.

3.3 Java I/O package

Input and output are one of the more uninteresting parts of any programming language. Every language has some facilities for it and the differences between systems tend to be more annoying than valuable. Sometimes the support is simple and pretty basic, other times it is full-featured and complex. My basic philosophy about language I/O is to learn it on the "need to know" basis-why pre-emptively invest a lot of time figuring all the variations of the different options to `printf`

when you may never even need to use them? Just look it up when you need to know. In that vein, we won't spend much lecture discussing Java's I/O system. You'll be on your own to read up in the Java package documentation, the reader, and Holmes to see what is out there. We will try to make the I/O portions of the assignment really pretty straightforward so they shouldn't cause you much grief.

Java falls somewhere toward the more complicated end of the spectrum of I/O systems. Although the java.io package contains rich and extensive support for all sorts of I/O needs, it can be unexpectedly difficult to accomplish simple things because you need to become familiar with many different classes. Things are further complicated by the fact that between versions 1.0 and 1.1, there were some significant re-structuring of the classes and their hierarchy. We also have provided the ucb.io libraries which are more straightforward for C programming refugees.

For this assignment, you will need to be able to break up a stream of text into its words (or parse it into tokens, to use the CS terms). The java.io.StreamTokenizer is just the sort of tool that is needed to do this. Rather than have you manipulate a tokenizer directly this time, we've made it a bit easier for you by providing a really simple Scanner class that configures and manages the tokenizer without requiring you to get too involved. One thing that is particularly nice about our Scanner is that it is aware of HTML formatting conventions and will extract an entire html tag as one token (rather than breaking it up into its component pieces).

When a new scanner is constructed, the client provides the input stream to scan (which can have come from an opened file, a URL connection, a stream created on a string, etc.). The client repeatedly messages the Scanner to getNextToken to retrieve tokens one by one. The method returns null when no more tokens exist which tells you that you have reached the end of the input.

The Scanner also comes with one bonus static method `findURLInTag` that is used to pull out the URL inside an anchor tag.

3.4 Threads & Synchronization

As noted in the description of the program, indexing goes on "in the background". The user is allowed to make searches on the index right while it is growing. To accomplish this concurrency, for each page to be indexed (including the first one), you should start up a separate thread to handle opening a connection to the site, retrieving the contents of the page, tokenizing the stream into words, and adding the references into the index.

The threads operate pretty much independently of each other, but they are sharing access to some of the same data which means you will need to take some care. One of the first things you will need to do is to add proper use of the synchronized keyword to the appropriate methods in your Multiset class so it will avoid problems when being accessed from more than one thread. You should also take a careful look at the other objects you are using (Vectors, URLs, Hashtable, 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 SiteSearcher class as well to avoid any critical sequences of calls being interrupted or interleaved in inappropriate ways.

4 Suggested Plan of Attack

Remember that your best strategy for handling a complex program is to proceed in stages and thoroughly test each piece before moving on. You are really handicapping yourself if you try to write the entire program and then debug the whole thing at once. Here is a sketch of a sequence of steps we might recommend you take to complete the assignment:

1. Check out the java.util.Vector class and see how to access and search for items.

2. Implement the basic operations detailed in Section 2.1.1. Test them thoroughly. Trust me, if your the basic operations in Multiset don't work, the whole project could get ugly
3. Implement the callback methods for the Multiset. The number of lines to write will not be very many but it requires a good understanding of interfaces.
4. Check out java.net.URL and URLConnection, figure out how to create a URL, determine its content-type and echo its contents to System.out
5. Take a look at our Scanner class and write a loop which takes an input stream and breaks it up into words and prints them
6. Organize and store the information about the words found in an input stream, using the Hashtable and Multiset, write a lookup loop to search the index and print results
7. Extract URLs found along the way in indexing and if they meet the restriction criteria, add them to the list to be indexed
8. Sequentially process all the URLs from the list and add their references into index
9. Plan out your threading strategy, add synchronized where needed to Multiset and SiteSearcher, move the indexing tasks to separate threads

5 Deliverables

The directory `~cs61b/hw/proj2` contains a few skeleton files that may suggest some structure for this project. Copy them into a fresh directory as a starting point. You can implement the methods as you wish and add whatever classes you want.

Starting with this assignment, we'll be grading your programs on more than just correctness. You will also expected to sufficiently document your code and have "readable" code. You are required to turn in a `README` file. The README should give an overview for every file you submit including abstraction comments which describe the purpose of the class. Use the handout on Achieving Readability in the reader as a guide for writing your overview and for commenting your actual code.

To submit your result, use the command '`submit proj2`'. Each partnership should turn in exactly one project (from either partner). Make *sure* that both of your names and logins are on everything you turn in. You will turn in nothing on paper.

Be sure to include tests of your program (yes, that is part of the grade). You'll especially need to extend the tests on the Multiset portion of the assignment.

6 Random Details

- Please do not change any of the class names, interface names, method names, or parameter lists to the public methods we specify above. This is to make it more straight-forward for us to test your code in an automated manner. Code which does not precisely match the specification will be heavily frowned upon.
- Take care to avoid duplicating code. There is quite a bit of repetition among the similar methods and you should take care to unify the common paths and share that code rather than repeat it in multiple places.

- For testing, you can start by accessing the cs61b page at `~cs61b/public_html/index.html`. You can access this via direct file access (if you are running on the local machines) with this URL `file:/~cs61b/public_html/index.html`. You can also copy the files to your local host and adjust the path in the URL to refer to your copy. When you are ready to test via network, you can access the same pages using `http://www-inst.eecs.berkeley.edu/~cs61b/index.html`. During testing, it is probably most convenient to temporarily hard-wire the starting URL and limits rather than having to type them in on each run.

- Consideration of others and conservation of shared resources means you should NOT repeatedly perform large searches of network sites. We strongly encourage you to do most of your testing by searching local file URLs so that you are only accessing files on your machine. Please limit yourself to only a few small searches over the network when necessary. Everyone who shares the bandwidth thanks you in advance for your proper network etiquette.

- When testing your program, we advise you to use conservative limits (say 25) on the maximum number of URLs to index, given that many Java systems have upper bounds on the number of concurrent threads and open files and connections. You might be able to go a bit higher, but no reason to push it.

- A Java program doesn't terminate until all threads finish. If the user wants to quit while threads are still exiting, you can forcibly exit by using the call `System.exit(0)` which immediately halts execution and exits the virtual machine.

- ***Start soon!!!*** This project is involved and requires a significant amount of coding. Fortunately, we have provided a decent road map of what needs to be done, but it could become an insurmountable of work if embarked upon at the last minute.