

HW4 Threads

Original Problem Concepts and Some Sections by Nick Parlante

Homework 4 gives us an opportunity to practice using Java Threads. The first programs also give you a chance to run programs from the command prompt. This assignment is due midnight of the evening of Thursday, February 10th. **Reminder:** This is the last assignment you may use late days for.

For those who haven't run java from the command prompt yet, here is a brief overview. To run a Java program from the command prompt, bring up a console window on your computer. Change directory to where the Java bytecode for your Java files is located.¹ Run the `public static void main` method for one of your classes by typing `java` followed by the name of the class followed by any arguments you want to pass to the main method. For example:

```
> java Bank small.txt 1
```

will execute the `public static void main` method from the class named `Bank`. The arguments `"small.txt"` and `"1"` will be passed to the main method and can be accessed through the `args` parameter as `args[0]` and `args[1]`. All arguments received will be strings. If you want to convert them to integers, you can use something like the `Integer` class's `parseInt` method.

Eclipse can simulate running from the command prompt. If you right-mouse on a class's name go to the "Run As" menu and choose "Run Configurations". Switch to the second tab in the dialog box which comes up and type whatever arguments you want to pass to your program.

Now on to this assignment's problems:

Bank

Multiple threads are a natural fit for programs involving producing and consuming items. One set of threads can be assigned to producing items while a separate set of threads can handle consumption. Solving producer/consumer type problems with multiple threads leads to an efficient use of resources. Multi-threading works particularly well if producing threads and consuming threads require different resources—for example if producing items is disk-intensive, whereas consuming items is CPU-bound.

Problem

For this problem we will simulate a bank. Our bank will track the balances in twenty different accounts. When the program begins, each of the accounts contains \$1000. The

¹ Java source files (*.java) compile into Java bytecode (*.class). If you create and edit your files in Eclipse, the *.java files should be in a subdirectory labeled `src` and the *.class files will be in a separate subdirectory called `bin`.

program will process a list of transactions which transfer money between the accounts. Once all transactions have been processed the program will go through and for each account it will print both the account's final balance and the number of transactions (deposits and withdrawals) which occurred on that account.

Our bank program will use the main thread to read the list of banking transactions from a file. A separate set of worker threads will process the transactions and update the accounts. Java 1.5's `BlockingQueue` will handle most of the communication between the main thread and the worker threads.

Instead of using a GUI your program will run from the command line. When executing the program users will provide two pieces of information as parameters—the name of an external file listing all the transactions and the number of threads which will be used to process transactions. Here is an example showing a request to process the transactions in the file “small.txt” using four worker threads.

```
> java Bank small.txt 4
acct:0 bal:999 trans:1
acct:1 bal:1001 trans:1
acct:2 bal:999 trans:1
acct:3 bal:1001 trans:1
acct:4 bal:999 trans:1
acct:5 bal:1001 trans:1
acct:6 bal:999 trans:1
acct:7 bal:1001 trans:1
acct:8 bal:999 trans:1
acct:9 bal:1001 trans:1
acct:10 bal:999 trans:1
acct:11 bal:1001 trans:1
acct:12 bal:999 trans:1
acct:13 bal:1001 trans:1
acct:14 bal:999 trans:1
acct:15 bal:1001 trans:1
acct:16 bal:999 trans:1
acct:17 bal:1001 trans:1
acct:18 bal:999 trans:1
acct:19 bal:1001 trans:1
```

Details

I recommend a design with four classes—`Bank`, `Account`, `Transaction`, and `Worker`. Both the `Account` and `Transactions` classes are quite simple.

- **Account** needs to store an id number, the current balance for the account, and the number of transactions that have occurred on the account. Remember that multiple worker threads may be accessing an account simultaneously and you must ensure that they cannot corrupt its data. You may also want to override the `toString` method to handle printing of account information.

- **Transaction** is a simple class that stores information on each transaction (see below for more information about each transaction). If you're careful you can treat the Transaction as immutable. This means that you do not have to worry about multiple threads accessing it. Remember an immutable object's values never change, therefore its values are not subject to corruption in a concurrent environment.
- The **Bank** class maintains a list of accounts and the `BlockingQueue` used to communicate between the main thread and the worker threads. The Bank is also responsible for starting up the worker threads, reading transactions from the file, and printing out all the account values when everything is done. **Note:** make sure you start up all the worker threads before reading the transactions from the file.
- I recommend making the **Worker** class is an inner class of the Bank class. This way it gets easy access to the list of accounts and the queue used for communication. Workers should check the queue for transactions. If they find a transaction they should process it. If the queue is empty, they will wait for the Bank class to read in another transaction (you'll get this behavior for free by using a `BlockingQueue`). Workers terminate when all the transactions have been processed. We'll discuss ways to communicate this below.

File Format

Each line in the external file represents a single transaction, and contains three numbers: the id of the account from which the money is being transferred, the id of the account to which the money is going, and the amount of money. For example the line:

```
17 6 104
```

indicates that \$104 is being transferred from Account #17 to Account #6.

Communication Mechanisms

Our blocking queue is the primary means of communicating between worker threads and the main bank thread. Worker threads requesting transactions from the queue will automatically block if the queue is empty, and will wait for the bank thread to add additional transactions. Similarly if the queue is full, the bank thread will automatically block until a worker thread takes a transaction out of the queue, making space to put the next transaction into. You can use a variety of classes to implement the blocking queue (remember `BlockingQueue` itself is an interface not a class)—the `ArrayBlockingQueue` will work fine for our purposes.

We need a way of communicating to the Worker threads when the main thread has finished loading all the transactions into the queue. Since the Worker threads are already getting information from the queue, one easy way to do this is to put a special value into the queue, indicating the main thread is done loading transactions. We can't use `null` because the `BlockingQueue` already uses `null` as a special value for its own communication purposes. Instead you can create a special transaction—something like this:

```
private final Transaction nullTrans = new Transaction(-1,0,0);
```

When your main thread is done reading in all the transactions, it places a series of `nullTrans` references into the queue. When the worker thread pulls a `nullTrans` out of the queue it knows that all the real transactions are done and it can terminate. Remember that the worker queues are actually removing these references from the queue, so you'll need to add one per worker queue.

You'll also need a way for the worker threads to let the main thread know when they are done. The main thread can't print out all the account balances until it's sure that all transactions have not only been entered in the queue but actually been processed. One easy way to do this is with a `CountDownLatch`.

Basic Testing

We've provided you with three files for testing purposes—`small.txt`, `5k.txt`, and `100k.txt`. The correct output for `small.txt` is shown above. For both `5k.txt` and `100k.txt` after all the transactions have been processed all accounts should be back to their original \$1000 balance level.

Hash Cracker

Hash functions are widely used in cryptography. One common use of Cryptographic Hash Functions is to store password information. Computer systems do not store passwords in “clear text” because any intruder finding the password file would instantly have a list of all the passwords to all the accounts in the system. Instead passwords are passed through a hash function and converted to hash values; these hash values are stored in the password file in lieu of the actual passwords. When a user logs in, the password they enter is run through the hash function and the result is compared to the hash value for the account in the file of password hash values. If the two hash values are the same, the user's identity is verified.

Here is an example using SHA, a very popular hash function. I set my password to “molly” which generates the hash value:

```
4181eecbd7a755d19fdf73887c54837cbecf63fd
```

My new password “molly” is not stored anywhere on the system. Instead the hash value is stored in the password file. Later, someone tries to log in to my account and guesses that my password is “flomo.” But “flomo” doesn't generate the same hash value as “molly”. Instead it generates the hash value:

```
886ffd41c568469795a19f52486bdde64f5f5bcc
```

The system compares the two hash values and determines that they are not the same and therefore concludes that the person logging in is not me.

In order for a hash function to work well in cryptography, it needs to have some important characteristics. Chief among them is that while it is easy to go from a password to a hash

value, it must be extremely difficult to go from the hash value back to the string or password which generated that hash value.

Suppose a hacker gets a hold of the password file and determines what the hash value for my password is. If the cryptographic hash function used to generate the hash function has no weaknesses, the only way for a hacker to determine the password is to perform a brute force attack. In a brute force attack the hacker generates all the possible passwords until they find the password that generates the hash value in the file.

Because of the computationally intensive nature of a brute-force password attack, threads are a natural part of any attack program.

Problem

For this assignment you will write `Cracker.java`, a program with a command-line interface. This program will runs in two modes, with the number of arguments listed on the command-line distinguishing the modes:

- In Generation Mode your program will take a password and will print out the corresponding Hash Value. For this simple task only the main thread will be used. Here is what the program looks like in this mode:

```
> java Cracker molly
4181eecbd7a755d19fdf73887c54837cbecf63fd
```

- In Cracking Mode your program will take a Hash Value, the maximum length of the password used to create the Hash Value, and the number of threads to use. The program will generate worker threads and use them to discover all passwords which might have the corresponding hash value.² Here is what the program looks like when executing. I have requested that it find the corresponding password, where the password length is limited to 5 and the program should use 8 threads.

```
> java Cracker 4181eecbd7a755d19fdf73887c54837cbecf63fd 5 8
molly
all done
```

Details

Passwords

To keep the number of passwords manageable we will be limiting the passwords which are allowed to those which can be formed from the following character set:

```
public static final char[] CHARS =
```

² While unlikely, it is possible for two or more passwords to generate the same hash values.

```
"abcdefghijklmnopqrstuvwxyz0123456789.,-!".toCharArray();
```

Here is the sequence of strings length two or less that we can make from those chars ...

```
a
aa
ab
ac
...
a-
a!
b
ba
bb
bc
...
b-
b!
c
ca
cb
...
!,
!-
!!
```

Threading

In Cracking Mode your program will spread the work across multiple threads. Each thread will be responsible for searching through some part of the input string space. Since each worker has its own, isolated part of the input space, the workers do not need to coordinate with each other once they are running.

Our strategy will be to give each worker a different part of the input space to run through, based on the first character of the strings.

- If there is 1 worker thread, then it uses all the input chars, a..! (indexes 0..39 in the CHARS string) at the start of its strings.
- If there are 2 worker threads, then the first worker creates all the strings starting with chars a..t (indexes 0..19), and the second worker creates all the strings starting with chars u..! (indexes 20..39).

- Do not first generate a collection of all possible inputs, and then feed them to the workers. The workers should do their own generation of inputs, so the generation happens in parallel too.
- There are 40 chars, and so which worker gets which starting chars may not divide up exactly evenly. That's ok, just so long as every starting char is accounted for exactly once, and all the workers have roughly the same number of starting chars. If the user really cares that things work out evenly, they can give a number of workers that divides into 40 evenly, such as 4 or 8 or 10 or 20.
- Assume a maximum of 40 worker threads.
- Probably the easiest way to think about the starting chars, is that each worker has a range of index numbers (e.g. 0..19, or 20..39), and it uses the chars from the CHARS array with those indexes.

Note that `System.out.println()` can be called by multiple concurrent threads and it does the right thing.

You'll want to print a message when all threads have completed their work. I recommend using a `CountDownLatch` to coordinate the main thread with the worker threads.

Generating the Hash

The Java `MessageDigest` class in the `java.security` package is designed to do Cryptographic Hash Functions. Check the online documentation for information on how it is used. Here are a few important notes for using it on our assignment.

- We will use the SHA algorithm (just as the example on the `MessageDigest` javadoc page does).
- Creating a new `MessageDigest` will require you to use a try-catch block. You can just call `e.printStackTrace()` in the catch section.
- The `MessageDigest` methods take `byte[]` as parameters and also generate `byte[]` as results. In Java the byte data type is a special scalar type which allows us to store number between -128 and 127. For converting String passwords to `byte[]` for the `MessageDigest`, you can simply call the String's `getBytes` method.

The `Cracker.java` start file includes a function to convert from `hexToArray` method to convert from Strings to `byte[]` and a `hexToString` to convert from `byte[]` to String. The `hexToArray` method can be used to converting the hash entered at the command line (passed in as one of your public static void `main`'s args) to a byte array. You'll use `hexToString` to convert the output of the `MessageDigest`'s `digest` method to a printable form.

- The `Arrays` class has an `equals` method which will allow you to compare two `byte[]` arrays. Note that is the `Arrays` class (with an 's' on the end).

Strategy and Basic Testing

I recommend getting the Generation Mode up and running first. This will give you a chance to make sure you understand how the MessageDigest class works and how to use the hexToArray and hexToString utility methods. It will also let you generate test cases for the Cracker Mode.

Start CrackerMode by running tests on short 1 or 2 character passwords. Be aware that the program can take quite some time if you increase the password length you are allowing much past 4 — even on relatively fast personal computers.

Here are some test Hash Values to get you started:

```
a is 86f7e437faa5a7fce15d1ddcb9eaeaea377667b8
fm is adeb6f2a18fe33af368d91b09587b68e3abcb9a7
```

The Count

As you may recall there are several important principles we need to follow when writing Swing applications:

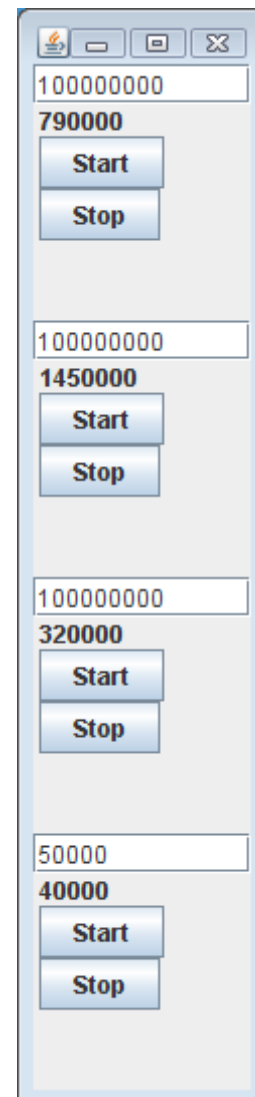
- All Swing methods (with a few specific exceptions) must be called from the Swing thread. We can ensure we don't violate this rule by making sure that all Swing calls are either already on the Swing Thread (e.g., Event Handlers which are always executed by the Swing Thread) or are passed to `SwingUtilities.invokeLater` for invocation.
- We must make sure that Event Handlers or other code executed on the Swing Thread does not “hog” the Swing Thread. When the Swing Thread is executing computationally expensive code, all event handling and painting ceases and the application will appear dead to the user. We can ensure that this does not happen by forking computationally expensive operations to a separate worker thread.

In this problem we explore these issues further.

Problem

We will create a user interface which allows us to create up to four independent worker threads. These worker threads will all count from 1 up to 100,000,000 (or any number provided by the user which fits in a standard int variable). At regular intervals the worker threads will update the user interface. Our interface is shown at right.

As you can see the interface includes four identical sections each consisting of a text field, a label, and start and stop buttons. Each



section works independently and can have its own independent counting thread. The text field stores the value toward which the worker threads are counting. The label represents the current value of the worker thread (only updated at intervals of 10,000). The Start button checks to see if an existing counting thread is running and if it is, it interrupts it, it then starts up a new counting thread (starting the count back at 0). The Stop button checks to see if an existing thread is running and if it is, it interrupts it. Interrupted worker threads should end without completing their counting work.

Details

Create a `JCount` class based on `JPanel`. This class will be used to represent each independent counting section. In addition `JCount`'s public static void `main` will start your application going by creating the frame and add four `JCount` panels into it. Use a `BoxLayout` as the layout manager for the frame.³

Now that we know how threads work, you should create your Java interfaces properly using `SwingUtilities.invokeLater` instead of creating your Java components directly in the main thread. Here is the standard template for creating a GUI on the Swing Thread:

```
public class JCount extends JPanel {

    private static void createAndShowGUI() {
        // create your GUI HERE
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

Each `JCount` should include the text field, label, buttons, and any internal variables needed to maintain its count. As with the outer frame, use `BoxLayout` as the layout manager for each `JCount`. To get the separation shown between the panels add a 40-pixel section to each `JCount` by creating a rigid area like this:⁴

³ `BoxLayout` takes the container which you are laying out as a parameter. As I mentioned in the Swing lecture a few weeks ago, frames actually contain an inner content pane, which we can usually ignore. In this case, the `BoxLayout` wants the inner content pane, not the frame passed in as a parameter—you can get to the content pane by calling `getContentPane()` on the frame.

⁴ For more information see Java Tutorial's page on Box Layout. The best way to get there is to Google "Java Layout Managers Visual Guide", then click through to the Box Layout tutorial. If you plan to use Layout Managers, get familiar with Sun's Visual Guide to Layout Managers.

```
add(Box.createRigidArea(new Dimension(0,40)));
```

Define a `WorkerThread` which handles counting. The `WorkerThread` should start counting from 0 up to whatever number is in the text field when the thread is created. Every 10,000 iterations have the `WorkerThread` go to sleep for 100 milliseconds⁵ and then update the label to show the current count. Remember that while setting text on Text Fields is one of the few Swing-related actions that can be done on or off the Swing Thread, updating `JLabel` can only be done on the Swing Thread, so you'll need to use `SwingUtilities.invokeLater`. Make sure that your `WorkerThread` ends when the counting is completed or when it is interrupted.

Basic Testing

If done properly the counters should all run independently of each other, and the user interface should remain functional even when all four sections have worker threads counting.

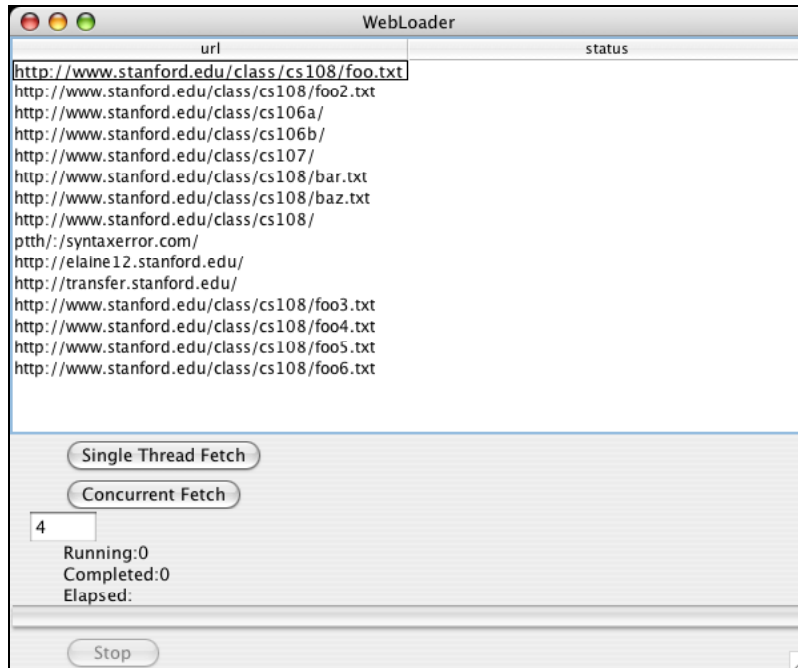
WebLoader

For this part, we will build a little URL downloading program that shows off the combined power of GUIs and threads. We will start with an overview of the operation of the program, and then talk about implementation strategies.

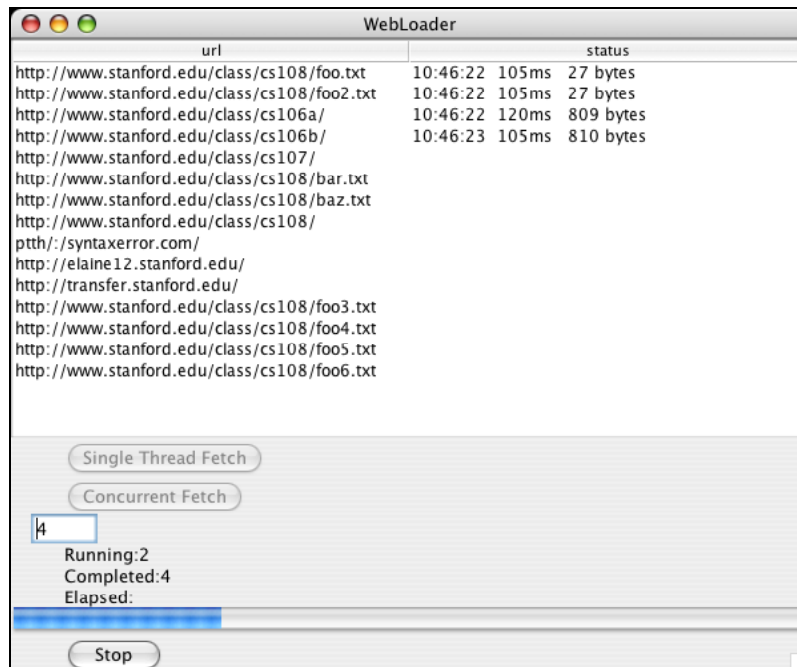
The `WebLoader` program loads a list url strings from a file. It presents the urls in the left column of a table. When one of the Fetch buttons is clicked, it forks off one or more threads to download the HTML for each url. A Stop button can kill off the downloading threads if desired. A progress bar and some other status fields show the progress of the downloads -- the number of worker threads that have completed their run, the number of threads currently running, and (when done running) the elapsed time.

Here is the `WebLoader` interface, with no workers running, ready to be started. The Fetch buttons are enabled, and the stop button is disabled (grayed out).

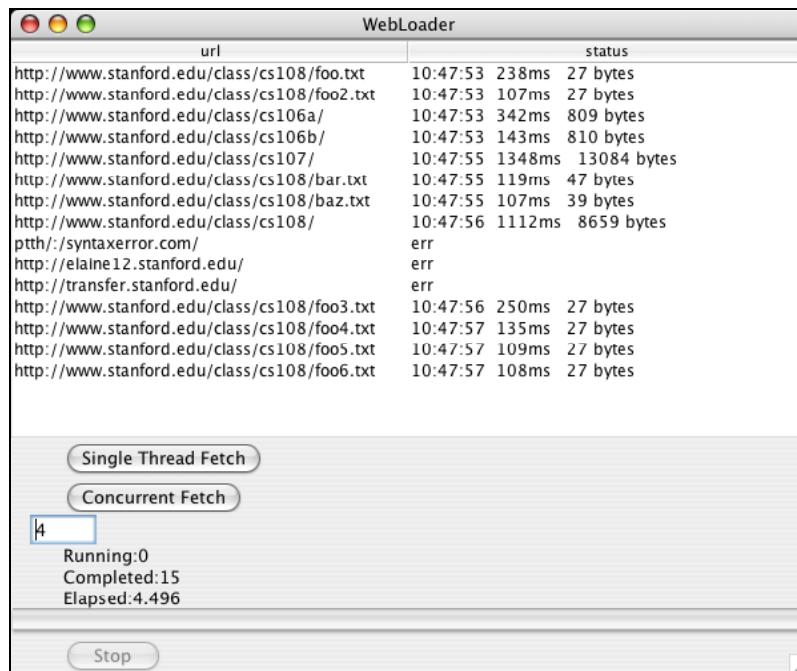
⁵ On a fast computer without the sleep call the counters will rise too quickly for us to experiment much with all four counters running.



Here, the Single Thread Fetch has been clicked, and one worker at a time, the program is proceeding through the urls. The Fetch buttons are disabled, and the Stop button is enabled. Four workers have finished and written their url's status. One worker is running. The "Running" thread count is 2, since it accounts for the one worker plus the one "launcher" thread described below.



Finally, here the fetch has been completed, with a run of 15 workers.



The screenshot shows a window titled "WebLoader" with a table of fetched URLs and their status. The table has three columns: "url", "time", and "status". Below the table are two buttons: "Single Thread Fetch" and "Concurrent Fetch". There is also a text input field containing the number "4", and a status section showing "Running:0", "Completed:15", and "Elapsed:4.496". At the bottom is a "Stop" button.

url	time	status
http://www.stanford.edu/class/cs108/foo.txt	10:47:53 238ms	27 bytes
http://www.stanford.edu/class/cs108/foo2.txt	10:47:53 107ms	27 bytes
http://www.stanford.edu/class/cs106a/	10:47:53 342ms	809 bytes
http://www.stanford.edu/class/cs106b/	10:47:53 143ms	810 bytes
http://www.stanford.edu/class/cs107/	10:47:55 1348ms	13084 bytes
http://www.stanford.edu/class/cs108/bar.txt	10:47:55 119ms	47 bytes
http://www.stanford.edu/class/cs108/baz.txt	10:47:55 107ms	39 bytes
http://www.stanford.edu/class/cs108/	10:47:56 1112ms	8659 bytes
ptth://syntaxerror.com/		err
http://elaine12.stanford.edu/		err
http://transfer.stanford.edu/		err
http://www.stanford.edu/class/cs108/foo3.txt	10:47:56 250ms	27 bytes
http://www.stanford.edu/class/cs108/foo4.txt	10:47:57 135ms	27 bytes
http://www.stanford.edu/class/cs108/foo5.txt	10:47:57 109ms	27 bytes
http://www.stanford.edu/class/cs108/foo6.txt	10:47:57 108ms	27 bytes

There are two main classes that make up the WebLoader program...

WebFrame

The WebFrame should contain the GUI, keep pointers to the main elements, and manage the overall program flow.

WebWorker

WebWorker is a subclass of Thread that downloads the content for one url. The "Fetch" buttons ultimately fork off a few WebWorkers.

The files "links.txt" and "links2.txt" in the starter directory have some urls for you to play with. As shown in the screen shots, some of the urls in links.txt should error out -- they are not actually web servers.

The starter files for this part are minimal -- WebWorker.java has the basic networking code shown below, and you should create your own WebFrame.

WebFrame Setup

The WebFrame constructor should read the file "links.txt" into a DefaultTableModel associated with a JTable installed in a panel, like this...

```
model = new DefaultTableModel(new String[] { "url", "status"}, 0);
table = new JTable(model);
table.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
```

```
JScrollPane scrollpane = new JScrollPane(table);
scrollpane.setPreferredSize(new Dimension(600,300));
panel.add(scrollpane);
```

Below the table, install the three buttons, three JLabels, one JTextField and one JProgressBar, with something like the appearance shown above. By default, the JTextField will grow to fill the width of the box, which works but does not look good. Use `setMaximumSize()` on the field to keep its size looking reasonable.

Fetching URLs

When one of the Fetch buttons is clicked, the GUI should change to a "running" state -- fetch buttons disabled, stop button enabled, status strings reset, maximum on the progress bar set to the number of urls.

The swing thread cannot wait around to launch all the workers, so create a special "launcher" thread to create and start all the workers. We will include the launcher thread in the count of "running" threads. Having a separate launcher thread helps keep the GUI snappy — we leave the GUI thread to service the GUI. Notice that GUI reacts quickly to mouse button clicks, etc. even as the launcher and all the worker threads are working and blocking.

The launcher should do the following

- Run a loop to create and start WebWorker objects, one for each url.
- Rather than starting all the workers at once, we will use a limit on the number of workers running at one time. This is a common technique -- starting a thousand threads at once can bog the system down. It's better to limit the number of concurrent workers to some reasonable number. When the Single Thread Fetch button is clicked, limit at one worker. If the Concurrent Fetch button is clicked, limit at the int value in the text field at the time of the click.
- Use a semaphore to enforce the limit -- the launcher should not create/start more workers than the limit, and each worker at the very end of its run can signal the launcher that it can go ahead and create/start another worker.
- At the very start of its run(), each worker should increment the running-threads count. At the very end of its run, each worker should decrement the running-threads count. The launcher thread should also do this to account for itself, so the running-threads count includes the launcher plus all the currently running workers.
- You can detect that the whole run is done when the running thread count gets back down to zero. When that finally happens, compute the elapsed time and switch the GUI back to the "ready" state -- Fetch buttons enabled, Stop button disabled, and progress bar value set to 0.

When the launcher/limit system is working, the running-threads status string should hover right at or below the limit value plus one (for the launcher) at first, and gradually go down to zero.

Make a new semaphore for each Fetch run -- that way you do not depend on the semaphore state left behind by the previous run. Interruption may mess up the internal state of the semaphore.

WebWorker Strategy

The WebWorker constructor should take a String url, the int row number in the table that it should update, and a pointer back to the WebFrame to send it messages.

The WebWorker should have a download() method, called from its run(), that tries to download the content of the url. The download() may or may not succeed for any number of reasons, it will probably take between a fraction of a second and a couple seconds to run

Given a url-string, Java has classes that make parsing the URL and retrieving its content pretty easy. The standard code to get an input stream from a URL and download its content is shown below (see the URLConnection API docs, and this code is in the starter file). The process may fail in many different ways at runtime, in which case the flow of control jumps down to the catch clauses and continues from there. If the download succeeds, control gets to the "Success" line.

```
InputStream input = null;
StringBuilder contents = null;
try {
    URL url = new URL(urlString);
    URLConnection connection = url.openConnection();

    // Set connect() to throw an IOException
    // if connection does not succeed in this many msec.
    connection.setConnectTimeout(5000);

    connection.connect();
    input = connection.getInputStream();

    BufferedReader reader = new BufferedReader(new InputStreamReader(input));

    char[] array = new char[1000];
    int len;
    contents = new StringBuilder(1000);
    while ((len = reader.read(array, 0, array.length)) > 0) {
        contents.append(array, 0, len);
        Thread.sleep(100);
    }

    // Successful download if we get here
}
// Otherwise control jumps to a catch...
catch(MalformedURLException ignored) {}
catch(InterruptedException exception) {
    // YOUR CODE HERE
```

```

        // deal with interruption
    }
    catch(IOException ignored) {}
    // "finally" clause, to close the input stream
    // in any case
    finally {
        try{
            if (input != null) input.close();
        }
        catch(IOException ignored) {}
    }
}

```

Things to notice...

- The code should test `isInterrupted()` periodically while reading and stop trying to read in that case. This will be important later as you try to get the Stop button to actually stop things. The worker may not notice `isInterrupted()` instantly, since the flow of control may be down in the `read()` library code for a time, before it makes it back out to your code to notice the interruption. The other possibility is that one of the blocking operations (e.g. `read()`) will throw `InterruptedException` and the flow of control will jump to that `catch()` clause.
- The `Thread.sleep(100)` line is a required slowdown for this assignment — otherwise it all happens too fast to interpret. I want you to see the progress and interaction of the threads, stop button, progress bar, etc., and slowing the threads down a little is the best way.
- The "finally" at the end runs whether or not there is an exception -- this is a standard use of the finally clause to be sure to `close()` the input.

When the download is done reading (successfully or not), the `WebWorker` should update the GUI.

First, the worker should deal with the corresponding status string in the table. If the worker was interrupted, it should set its status to "interrupted". Otherwise, if the worker was not interrupted it should update the status as follows: If the download was successful, update the status of the corresponding row in the table with a `String` that summarizes the download, giving the wallclock time of completion, the elapsed time of the download in milliseconds, and the size in bytes of the downloaded content (use the number of chars, although in reality the number of bytes could be larger for a page with non-ASCII content). See the `Date` class to figure the current time, and the `SimpleDateFormat` class to format the current time reasonably. If the download was not successful, just update the status to "err". We could do something with the content, but for this program we just download it and count how many chars there were (the length of `StringBuilder`).

Second, whether or not it was interrupted, the worker is about to exit and should update the GUI: decrease the count of running threads by 1, increase the count of completed threads and the progress bar by 1 (finishing for any reason -- interruption, success, err -- counts as "completion" for the thread), and work with the launcher semaphore to open up a slot for

another worker. We are not counting the launcher thread in the completed count, just the workers.

The completed and current-running counts should only be changed when actually launching a thread, or when a thread is actually at the end of its run(). Don't just set them to zero when you suspect all the threads are done -- let the exiting of the threads manipulate them on their own. That way, the GUI will give you an accurate view of how your thread lifecycle code is working.

Interruption

The Stop button should interrupt the launcher (so it stops launching new workers) and all the other workers. The threads should notice when they have been interrupted without too much delay. Depending on the JVM implementation, there may be a little delay before the workers get to a point where they notice they have been interrupted (if they are blocked down in read() or connect() for example. Windows I/O seems to be especially slow noticing interruption).

When interrupted, the launcher should stop creating and starting new worker threads. The launcher can just exit its run. Eventually all the other workers will notice that they have been interrupted, and the running count will go down to 0. Since the labels track the running and completion of threads, you can watch the GUI as you play with the Fetch and Stop buttons to see that your threads are starting up and closing down properly.

Tricky timing case: what if the user clicks the stop button just as the launcher is creating and starting another worker. You can imagine a pathological case where the Stop button interrupts all the workers just after the launcher checks isInterrupted(), so the new worker doesn't get interrupted. Solution: introduce some locking so that the task of creating a new worker and the task of interrupting the launcher and all the workers cannot run at the same time.

Here is a screenshot after a Concurrent Fetch that was interrupted with the Stop button. Looking at the table, we see that 13 of the 15 rows completed and wrote their status. Four workers were downloading when the Stop button was clicked. The interruption happened before the launcher ever got the chance to start a worker for two bottom rows.

WebLoader			
url	status		
<u>http://www.stanford.edu/class/cs108/foo.txt</u>	12:30:37	117ms	27 bytes
http://www.stanford.edu/class/cs108/foo2.txt	12:30:37	122ms	27 bytes
http://www.stanford.edu/class/cs106a/	12:30:37	119ms	809 bytes
http://www.stanford.edu/class/cs106b/	12:30:37	117ms	810 bytes
http://www.stanford.edu/class/cs107/	interrupted		
http://www.stanford.edu/class/cs108/bar.txt	12:30:37	355ms	47 bytes
http://www.stanford.edu/class/cs108/baz.txt	12:30:37	352ms	39 bytes
http://www.stanford.edu/class/cs108/	interrupted		
ptth://syntaxerror.com/	err		
http://elaine12.stanford.edu/	err		
http://transfer.stanford.edu/	err		
http://www.stanford.edu/class/cs108/foo3.txt	interrupted		
http://www.stanford.edu/class/cs108/foo4.txt	interrupted		
http://www.stanford.edu/class/cs108/foo5.txt			
http://www.stanford.edu/class/cs108/foo6.txt			

Running:0
Completed:13
Elapsed:0.597

Deliverables

Turn in all your HW4 project directory with a README as usual. Please delete the 100k.txt file before submitting -- no need to upload such a big file!