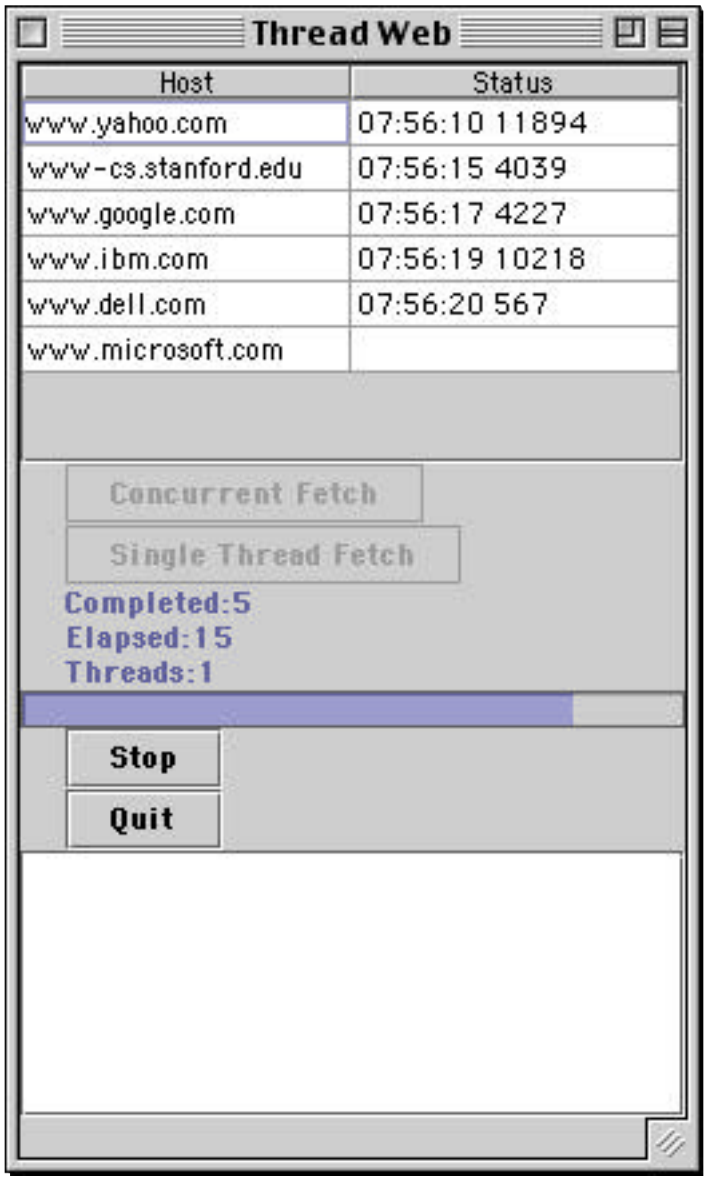
Handout #11
Nick Parlante

HW2c — ThreadWeb

Welcome to our last exercise in Java threading. If hw2b was a traditional, nuts-and-bolts threading exercise, hw2c is a super hip "Internet Age" threading extravaganza. For this assignment, you will build a little web client that happens to benefit greatly from Java threading.



All the parts of HW2 are due midnight ending Thu May 11th. There are some starter materials in the homeworks directory.

Here's an overview of how ThreadWeb works: Thread Web starts with a list URLs loaded from a file. It presents the URLs in a table. When one of Fetch buttons is clicked, it forks off one or more threads to download the HTML for each row. A progress bar and some other status fields show the progress of the downloads. A Stop button can kill off the downloading threads if desired.

Here are the classes which make up ThreadWeb...

WebRow

WebRow is the data model behind one row in the table. It stores the URL and the downloaded contents.

WebModel

WebModel is a subclass of AbstractTableModel that stores a Vector (or ArrayList) of WebRows and makes them look like a 2 column table — the hostname goes in column one and a status string describing the row goes in column two. The status string shows the time the download completed, and the number of bytes in the content.

WebWorker

WebWorker is a subclass of Thread that runs the download for a WebRow to get the latest version of its contents. Clicking the "Fetch" buttons basically forks off a few WebWorkers.

Semaphore

The classic counting Semaphore — it'll be used off the shelf to throttle the number of concurrent running WebWorkers to a reasonable level.

ThreadWeb

The root window of everything. Contains all the GUI elements and, for convenience, keeps pointers to everything else as needed such as the table model, the thread group, etc.

How It Works

The text below outlines how the pieces fit together. For things that aren't too interesting, the code is largely given to you (e.g. I/O). For other areas, the constraints on the solution are given but the code is up to you. You will also need to dig around in the Sun docs to see how to get the various off-the-shelf classes to work for you.

WebRow Strategy

The key method in WebRow is download() — a thread can come in on download() to try to download this row's HTML. The download() may or may not succeed for any number of reasons, it will probably take between a half a second and a couple seconds, and it will spend much of its time blocked waiting for its socket. If it succeeds, the contents of the row should be updated and the time should be noted.

Fortunately, the URL class and other built in Java classes make retrieving HTML given a URL pretty easy once you know the stream code which, surprise, we're giving to you...

```
URLConnection connection = null;
InputStream input = null;

try {
   connection = url.openConnection();
   connection.connect();
   input = connection.getInputStream();
   InputStream stream = connection.getInputStream();
   BufferedReader reader  = new BufferedReader(new InputStreamReader(stream));

   char[] buff = new char[1024];
   int len;
   StringBuffer result = new StringBuffer(8000);
   while ((len = reader.read(buff, 0, buff.length)) >0) {
      result.append(buff, 0, len);
      Thread.sleep(100);   // required slowdown
   }

   // Geting to here means success -- do something with result

}
catch(MalformedURLException ignored) {}   // Here are many ways to fail
catch(IOException ignored) {}
catch(InterruptedException e) { Thread.currentThread().interrupt(); }
finally {
   try{
      if (input != null) input.close();   // Subtle, reliable way to make sure
   }                                      // socket is released on the way out
   catch(IOException ignored) {}
}
```

Things to notice...

- There are many ways the process can break down — your solution may lump all the it-didn't-work cases together. In that case, the state of the WebRow should not change.
- The code should test Thread.currentThread().isInterrupted() periodically and bail out appropriately. This will be important later as you try to get the Stop button to actually stop things.
- 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. It also helps to cut down on the volume of Internet traffic we generate.

As in lecture, the catch clause of InterruptedException sends itself interrupt() so that subsequent checks of isInterrupted() will return true.

Note the use of the finally clause to make sure that the input gets closed no matter what. This is a classic use of the finally clause.

WebWorker Strategy

The constructor for WebWorker should take the WebRow to do the download() on, and the ThreadGroup that the thread should belong to. The WebWorker will need to store pointers to a few other things in its constructor for use later in run(). In its run(), the WebWorker should...

1. Inform the GUI that there is one more thread running, so the GUI can increment the "Thread" status line which counts the number of running download threads. This should be the first thing in run().

2. Attempt a download() on the web row

3. If the download is successful setRow() the row back into the data model (see below)

4. Check for isInterrupted() throughout.

5. Inform the GUI that there is one less download thread running just before exiting run(). The "Elapsed" label should increase to show the number of elapsed seconds since the Fetch button was clicked. Finally, if the download was successful, the progress bar and "completed" status count should advance by 1.

The screenshot on page 1 shows a download that is running in single-thread mode and has completed 5 of 6 rows in 15 seconds. You can tell it's running in single-thread mode since the times are strictly increasing in the rows as your read down. It's a little slow since I slowed it down to get a screen shot and my home Internet connection is not that fast anyway. When you have the concurrent case working, the times will reflect which rows came back first and will not just be top-down.

The WebWorker operates on a WebRow. However, the WebRow is still installed in the WebModel acting as the data store for a table, responding to getDataAt(), etc., so changing it while it's the data model for the table will violate the Swing thread constraint. What we really want it to run the download freely and then check the content back in once we've got it all. The solution is to run WebWorker with a temporary copy of the WebRow. The new, copied WebRow should have the same URL as the original but be separate otherwise. The copy allows the WebWorker to work concurrently and independently from the table model. WebRow() (below) will take care of re-integrating the row with the model after the row has been successfully downloaded.

WebModel Strategy

Much of the WebModel is standard TableModel behavior. It will also need accessors such as getRow(), addRow(), and setRow() to manipulate the rows. The getValueAt() should return the URL hostname as the first column, and for the description it should make up the status string...

Use the empty string if there are no contents.

Otherwise start with the hours:minutes:seconds of when the download completed. (see the SimpleDateFormat class).

Follow with the length in bytes of the contents.

The cells should not be editable

Finally, the WebModel should have a loadFile(File file) method to read the URLs line by line out of a file. The I/O looks like...

```
// The standard incantation to read a text file...
File file = new File(filename)
FileReader fileReader = new FileReader(file);
BufferedReader bufferedReader = new BufferedReader(fileReader);

String line;
while ((line = bufferedReader.readLine()) != null) {
   addRow(new WebRow(new URL(line)));
}
```

ThreadWeb

The ThreadWeb subclass of JFrame can arrange the GUI in a vertical box in the center of border layout. First there should be the table in a scroller. Give the scroller a preferred size of 200x150. After that should be the buttons, labels, and progress bar. Feel free to create a more artful layout if you wish.

The program should load its links from the file on the command line, or the file "links.txt" if there is no command line argument. I tried to use large sites so our burst of traffic Thu night will not cause me to be hunted down by a mob of angry ~~students~~ network administrators.

The tricky part of ThreadWeb is how it implements the Fetch buttons. Here's what a Fetch needs to do, ignoring the minor issues for a moment: Change the GUI to the "running" state — fetch buttons disabled, status strings reset, correct maximum on the progress bar. Create a thread group for all the new threads. Create a "launcher" thread to launch the worker threads (Pun: the launcher thread is the "Helen" thread). Having a separate launcher thread allows us to keep the GUI snappy — we leave the GUI thread to service the GUI. Notice that GUI reacts snappily to mouse button clicks, etc. even as the download threads are working and blocking.

Our strategy will be to allocate all the WebWorkers first (creation is cheap) and then be careful about how many we allow to be running at once (running is

expensive). The launcher thread should create one WebWorker for each row and store them all in an array. After they are all created, the launcher should iterate through them sending start() to each one (see "throttling" below). Finally the launcher needs to use join() to wait until all the threads are finished, and then change the GUI to the "done" state — (stop button disabled, fetch buttons enabled, progress bar at 0, status strings left as they were. The sto p button should disable only when all of the WebWorkers have exited. It turns out that join() succeeds on a thread that has not yet been started, so the launcher can be indescriminate with its joining.

There are just a couple other complications...

1. Start Throttling

We will have a limit to how many WebWorkers may be running at one time. For the concurrent fetch, there should be no more than 4 WebWorkers running at one time, although there may be any number allocated but not started. (You may experiment with higher values, but please turn the program in with limit of 4.) Use one instance of the classic counting semaphore to throttle back the rate that the .start()'s happen. Be careful where you place the incr() and decr() requests so that each thread that finishes its run() (no matter how!) opens up a slot for another WebWorker to get started. The "Single Thread Fetch" button just runs the same code but with the throttle value set to 1.

In our implementation of the classic semaphore, the call to decr() may return because you have obtained the resource, or it may return because you have. been interrupted — test which case you got after calling decr().

If the throttle is working, the number-of-threads status string should hover right at or below the throttle value at first, and gradually go down to zero.

The throttle is actually pretty realistic. Most workstations prefer a relatively small number of simultaneous sockets (4..30) optimally, and will bog down with less overall throughput when trying to do more than that at once. We're using a relatively small throttle value to help keep our network traffic down. There's also typically an operating system limit on the number of simultaneous open sockets. Without the throttle, the program would have serious problems given a links.txt with, say, a few thousand rows. With the throttle it works fine no matter how many URLs there are in links.txt.

You may want to 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 messes up the state of our general semaphore implementation.

2. Interruptions

There are many places to notice interruptions in this algorithm. WebWorkers should just gracefully exit on interruption. The launcher should be smart enough not to start new threads once interruption has started. The launcher thread should still wait for all the workers to exit before changing the GUI to the stop state — this will be your visual feedback that you have successfully interrupted all your threads.

3. The Stop Button

The Stop button can just do an interrupt() on the thread group. There may be a tiny lag between hitting the stop button and the die-off of the threads. If the progress bar or status keeps making progress a second after you hit the stop button, your threads are not noticing the interruption optimally. On some VMs, the read() is not necessarily unblocked by the interrupt() making the lag more irregular. However, even with the lag, interrupted threads should be smart enough to not advance the progress bar. Interrupted threads should still decrement the "running threads" label normally, so it should quickly drop towards 0 when you hit the stop button.

Optional HTML Pane

This last part it totally optional and irrelevant, but neat. In your ThreadWeb constructor, the following will add an HTML panel at the bottom of the frame...

```
editor = new JEditorPane("text/html", "");
editor.setEditable(false);
scrollpane = new JScrollPane(editor);
scrollpane.setPreferredSize(new Dimension(200, 100));
box.add(scrollpane);
ListSelectionModel lm = table.getSelectionModel();
lm.addListSelectionListener(this);
```

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

```
   public void valueChanged(ListSelectionEvent e) {
      int row = ((ListSelectionModel)e.getSource()).getAnchorSelectionIndex();
      if (row>=0) {
         // get webRow #row
         editor.setText(webRow.getContents().toString());

      }
   }
```

The HTML editor classes in Swing are a work in progress. It mostly renders many of the pages, but eventually blows itself up with exceptions and running out of memory. It will be interesting to run your code against JDK 1.3 to see if the HTML render has gotten any bettr. You can leave the fragile HTML feature in — we won't mess with it.

The Lesson Of Sockets And Threads

If years from now you remember nothing else from our little thread adventure, try to remember the following exercise. Run the program with the throttle value at 1. Then run with a throttle value of 4, and then with 8. What sort of speedup do you see for increasing the concurrency? How does this compare to the concurrency utilization in HW2b? Why is this? We'll discuss this in lecture. The basic idea is that concurrency and high latency activities go well together. The Internet is full of high latency activities.

Deliverables

Put all your hw2a, 2b, and 2c materials in one direcrory with a Readme. See the submit directions in the class directory for help with logistics. Please remove the .tr files from your directory before submitting

- We should be able to compile with "javac *.java". You should test your code on the elaines and sagas (dual processors).
- We should be able to run your 2a with "java MagicThread"
- We should be able to run your 2b with "java ThreadBank file.tr". It should run the transactions and then print out the balances of the 20 accounts one per line with the format *account_num balance*.
- We should be able to run your 2c with "java ThreadWeb links.txt". Please set your concurrent download to use a throttle of 4.