Com S 430
Spring 2018
Homework 2

General instructions

- When submitting modifications of existing code, **please do not reformat any sections of the code that you did not explicitly have to change**. We need to be able to diff your submission against the original to see exactly what was changed.
- Submit an archive on Canvas containing all classes originally posted as hw2.zip.

1. a) The package **`hw2.worker`** contains an application that is somewhat similar to the original client program from Homework 1. There is a GUI that keeps a local list of "staff information" that can be browsed. There is a box into which a name can be entered, and then the "add" button will use a very slow service to look up information for that name, and then add the information to the list. Two versions of the main class are included:

- **`ListDemo1`** does not use an auxiliary thread to do the lookup, and the GUI is therefore locked up for the duration.
- **`ListDemo2`** is an initial attempt to use an auxiliary thread, but it suffers from race conditions, as it violates the *single-thread rule* (see Section 9.1.2 of JCIP).

Modify **`ListDemo2`** to resolve the race conditions, using the existing inner class **`LookupWorker`**. (Remember that using the **`synchronized`** keyword for methods that access Swing components is generally never going to work.)

b) The **`hw2.function_plotter`** package is a simple program for graphing functions. The main logic is in the class **`FunctionPlotter`**, and there is a text-based user interface you can run in **`FunctionPlotterMain`**. The **`FunctionPlotter`** starts up a Swing application, **`GraphWindow`**, that displays a graph of the function. Take a look at the and try running it. It may work fine, or not, but it has some violations of the Swing single-thread rule. Modify the **`GraphWindow`** code to fix these problems. (You should not need to modify any of the other files.)

2. The class **`hw2.FileLogger`** is an attempt to implement a logging utility whose output goes to a text file. It is a terrible implementation. Every log message requires a disk access, so there would be a major impact on performance, and it isn't safe for use by multiple threads. Fix it using the following ideas:

- The log messages should go into a bounded thread-safe queue.
- A background thread reads the messages from the queue and writes them to the file.
- It is reasonable to expect the writers to block briefly if the queue is full, but the writers should not normally be blocked while the file is being written. (Note that the **`BlockingQueue`** interface includes a method **`drainTo`** that removes all elements of a queue in a single operation.)

You should not modify the public API for the logger.

3. a) Read the API for **`java.util.concurrent.CountDownLatch`**. The class javadoc includes a detailed example. Using the example given there as a guide, modify our **`hw2.Incrementer`** example to use a **`CountDownLatch`** to start and join the worker threads.

b) Implement your own **`hw2.CountDownLatch`** including just the two methods **`await`** and **`countDown`** from the API, using wait/notify/notifyAll. Modify the package declaration in your **`Incrementer`** to try it out.

4. The class **`hw2.list.LinkedList`** is a simplified singly-linked list implementation. There is also an interface for a simplified list iterator containing only methods **`hasNext()`**, **`next()`** and **`add(item)`**. There is no provision for removing elements, but new elements can be added via the list iterator. In this problem, we explore two ways to provide multiple threads with a consistent snapshot of the list while allowing traversals to occur concurrently and allowing the list to mutate and keeping copying to a minimum. Initially the classes **`ImmutableList`** and **`VersionedList`** are identical to **`hw2.list.LinkedList`** except for the renaming.

a) Modify **`ImmutableList`** for concurrent iteration by using immutable nodes. Once the iterator object is obtained, no further synchronization should be necessary for a thread to traverse the list, and the thread should see a valid snapshot of the list as it existed when the iterator was created. Since the links are immutable, in order to add an element, the preceding nodes of the list have to be *copied* (including the list's head node). Since multiple threads may try to **`add()`** at the same time, this operation needs to be done under synchronization (that is, calls to **`add()`** are atomic with respect to one another and the effects of one **`add()`** must be visible to the next **`add()`** and subsequently created iterators).

When a thread attempts to use an iterator's **`add(X)`** operation at position **`cursor`**, it must go back to the current head of the list in order to copy all nodes preceding **`cursor.next`**. Note that an iterator may not always be able to successfully add to the list. It may be the case that this iterator's **`cursor.next`** node is no longer reachable from the list's head. (This happens if some other iterator has updated a sufficiently large prefix of the list.) So, if the Node *instance* pointed to by this iterator's **`cursor.next`** no longer appears in the list, then we cannot add in an item to precede **`cursor.next`**. In this case, the **`add(X)`** attempt should fail by throwing a **`ConcurrentModificationException`**.

On the other hand, if (under synchronization) this iterator finds that the object instance pointed to by **`cursor.next`** *is* still reachable from the head, then our call to **`add(X)`** should succeed. By succeed, we mean that we can add a node with item X directly preceding **`cursor.next`**. Then make node copies going backwards from our newly added node back through **`head`**. Lastly, we update **`head`** so that subsequently created iterators will then begin by traversing our new node copies.

b) Modify **`VersionedList`** for concurrent iteration by using versioned iterators. This time, we want to leave the **`next`** links from one node to the next as mutable. Because these pointers may be mutated by iterators in different threads, accesses and updates by these iterators need to be synchronized. However, we don't want one iterator's accesses to unnecessarily block the progress of another iterator. We want threads to be able to iterate over the list concurrently. (Thus, each node needs its own lock.)

We also want an iterator to "view" the list as it was when the iterator was created. To make the iterator have a consistent "snapshot", make the iterator's **`next()`** method skip over any nodes that were added after the iterator was created. This is most easily done with version numbers: create each node with a unique, sequentially assigned ID number, and let the list's current version be the max ID number of any node it contains. To get our consistent snapshot, an iterator created when the list has version V should only "see" (i.e. return from **`next()`**) those nodes with IDs less than or equal to V.

A thread calling **`add()`** on its iterator can always (eventually) successfully insert an item directly after that iterator's current cursor. Thus, the **`add()`** operation does not ever need to throw a **`ConcurrentModificationException`**.