Com S 430
Spring 2015
Homework 2

Start by downloading the **`examples/week4/hw2`** and **`examples/week4/syntaxtree`** directories. You can also find examples of implementation and use of futures in **`examples/week4`**. All of the problems require you to make some modifications to existing code. Please leave the formatting alone for the parts of the code that you don't modify (that makes it easier to run a diff utility if we need to). You should submit a zip archive of the same files on Blackboard, including your modifications and comments.

Please start early. There are likely to be questions, ambiguities, errors, or other issues. Post questions on Piazza or talk with the staff.

1. The class **`FileLogger`** is an attempt to implement a logging utility whose output goes to a text file. It is a horrible 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 go into a bounded thread-safe queue, and a background thread reads the messages from the queue and writes them to the file. It is reasonable to expect the writers to block if the queue is full, but the writers should not normally be blocked while the file is being written. Add a comment explaining your changes.

Optional: The number of I/O operations can be minimized by having the reader concatenate all the log messages and open the file just once to write them. Come up with an implementation in which the reader only acts when a) the queue becomes full, or b) every 15 seconds, whichever comes first.

2. Read the API for **`java.util.concurrent.CountDownLatch`**. Using the given example as a guide, modify our Incrementer example to use a **`CountDownLatch`** to start and join the worker threads.

3. Come up with an implementation of the **`CountDownLatch`** API using wait/notify (you only need implement **`await`** and **`countDown`**).

4. (*Based on a true story*.) An IDE such as Eclipse includes a feature that will provide an "outline view" of a source file when a node in the explorer pane is clicked. The outline is based on a model which is constructed from the abstract syntax tree produced by a parser. Similarly, the syntax tree might have to be constructed if the file is opened for editing. Constructing the parser and performing the parse are relatively expensive operations, so the syntax tree is only constructed the first time it is needed. Upon construction, the syntax tree is kept in a cache for reuse until the file is modified. (In real life, this has to be a least-recently-used cache of bounded size, but we can ignore that issue for now.)

Note that various threads may need access to the syntax tree, (e.g. the editor or the event handler for expanding the outline view) so the cache has to be thread-safe. The syntax trees themselves can be considered to be effectively immutable.

The sample code **`syntaxtree.SyntaxTreeManager`** is a bare-bones implementation of a class for managing the store of already-constructed syntax trees. It suffers from the major defect that if one thread needs the tree for a file f that is *already* parsed while another thread is busy parsing a different file g, the first thread is blocked until the parse of g is finished.

Modify the implementation to address the following concerns (approximately in increasing order of difficulty). See **`syntaxtree/Test.java`** for a bit of test code.

1. Allow access to the cache and parsing to run concurrently, in the sense that a thread may get an existing syntax tree or initiate construction of a new one while other threads are parsing files.

2. Make sure that two threads don't parse the same file at the same time – if one thread is already parsing a file, the second thread should block until the first thread completes, and then return the result.

3. Since getting a syntax tree is a now a potentially blocking operation, make sure the operation is interruptible. (This might be a good opportunity to take another look at the idea of a Future.)

Add a comment to the top of your **`SyntaxTreeManager`** file explaining your design.