Concurrency: Building Thread Safe Data Structures

In your recitation directory we have provided: (1) a `edu.cmu.cs.cs214.rec11.queue` package with a `SimpleQueue` interface and an incomplete queue implementation, (2) a `edu.cmu.cs.cs214.rec11.map` package with a `SimpleHashMap` class, and (3) various test files in a corresponding package in the `test` directory. Your task is to use primitive Java synchronization to implement a correct unbounded blocking queue from our incomplete implementation and to fix the race conditions in the `SimpleHashMap`, which is not currently thread safe.

Implementing a blocking queue

An *unbounded blocking queue* is a normal queue except, if the queue is empty upon a dequeue request, then the request is blocked until an element is enqueued in another thread. This behavior contrasts with the standard `java.util` queue implementations, which either return `null` or throw a `NoSuchElementException` if the queue is empty. In an unbounded blocking queue the dequeue method will always return a valid element from the queue, although the dequeuing thread might wait arbitrarily long to dequeue an element.

The `edu.cmu.cs.cs214.rec11.queue.UnboundedBlockingQueue` is not yet thread safe; if multiple threads access the same queue concurrently, race conditions can occur and you might obtain unexpected results.

To complete this part of the recitation you should:

1. In the `src/test/main` folder, we have provided a `edu.cmu.cs.cs214.rec11.queue` test package that tests for the desired behavior of an unbounded blocking queue. Run the tests and understand their expected behavior. These tests will initially fail because the `UnboundedBlockingQueue`, as given, is not thread safe and does not block when a thread attempts to dequeue from an empty queue.
2. Using basic Java synchronization and the `wait` and `notify` methods, eliminate the race conditions in the `UnboundedBlockingQueue` and make it a correct unbounded blocking queue. In other words, enqueueing an element should always succeed immediately. An attempt to dequeue from an empty queue, however, should block until an element has been enqueued by another thread. To simplify your implementation, prevent race conditions by allowing only one thread to enqueue or dequeue at a time. Use the provided JUnit tests to evaluate the correctness of your implementation. Look at the sample concurrency code from lecture for more details.

Implementing a thread safe concurrent hash map

In the `edu.cmu.cs.cs214.rec11.map` package we have provided a `SimpleHashMap` class that is currently not thread safe. Your task is to use primitive Java synchronization to make the implementation thread safe. Specifically:

1. Discuss how you could use multiple locks to allow multiple threads to use the hash map concurrently, and discuss the trade-offs of coarse-grained (e.g., one lock) vs. fine-grained (multiple locks) synchronization strategies.
2. As time permits, implement a fine-grained locking strategy for the `SimpleHashMap`. In what way does our implementation simplify the use of a fine-grained locking strategy?

Annotations for Concurrency

You'll notice that there are a few annotations scattered throughout the code, such as `@GuardedBy` and `@ThreadSafe`. These are purely for documentation purposes and do not affect the code at runtime or compile-time. However, it is possible for tools (similar to FindBugs) to be developed to do static analyses, taking these annotations into account. See http://jcip.net/annotations/doc/index.html for the documentation.