Reentrant Lock#

Java's answer to the traditional mutex is the reentrant lock, which comes with additional bells and whistles. It is similar to the implicit monitor lock accessed when using `synchronized` methods or blocks. With the reentrant lock, you are free to lock and unlock it in different methods **but not with** different threads. If you attempt to unlock a reentrant lock object by a thread which didn't lock it initially, you'll get an **IllegalMonitorStateException**. This behavior is similar to when a thread attempts to unlock a pthread mutex.

Condition Variable#

We saw how each java object exposes the three methods, `wait()`, `notify()` and `notifyAll()` which can be used to suspend threads till some condition becomes true. You can think of Condition as factoring out these three methods of the object monitor into separate objects so that there can be multiple wait-sets per object. As a reentrant lock replaces `synchronized` blocks or methods, a condition replaces the object monitor methods. In the same vein, one can't invoke the condition variable's methods without acquiring the associated lock, just like one can't wait on an object's monitor without synchronizing on the object first. In fact, a reentrant lock exposes an API to create new condition variables, like so:

```
Lock lock = new ReentrantLock();
Condition myCondition  = lock.newCondition();
```

Notice, how we can now have multiple condition variables associated with the same lock. In the `synchronized` paradigm, we could only have one wait-set associated with each object.

java.util.concurrent#

Java's util.concurrent package provides several classes that can be used for solving everyday concurrency problems and should always be preferred than reinventing the wheel. Its offerings include thread-safe data structures such as `ConcurrentHashMap`.