Assignment5

Instead of using implicit locking via the synchronized keyword the Concurrency API supports various explicit locks specified by the Lock interface. Locks support various methods for finer grained lock control thus are more expressive than implicit monitors.

Multiple lock implementations are available in the standard JDK which will be demonstrated in the following sections.

public void lock(): Acquires the lock.

public void lockInterruptibly() “Acquires the lock unless the current thread is interrupted.

public Condition newCondition(): Returns a new Condition instance that is bound to this Lock instance.

public boolean tryLock(): Acquires the lock only if it is free at the time of invocation.

public boolean tryLock(long time, TimeUnit unit): Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.

public void unlock(): Releases the lock.

A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive.

readLock(): Returns the lock used for reading.
writeLock(): Returns the lock used for writing.

ReentrantReadWriteLock class of Java is an implementation of ReadWriteLock, that also supports ReentrantLock functionality.

The class ReentrantLock is a mutual exclusion lock with the same basic behavior as the implicit monitors accessed via the synchronized keyword but with extended capabilities. As the name suggests this lock implements reentrant characteristics just as implicit monitors.

The interface ReadWriteLock specifies another type of lock maintaining a pair of locks for read and write access. The idea behind read-write locks is that it’s usually

safe to read mutable variables concurrently as long as nobody is writing to this variable.

CompletableFuture is used for asynchronous programming in Java. Asynchronous programming is a means of writing non-blocking code by running a task on a separate thread than the main application thread and notifying the main thread about its progress, completion or failure.

A Future is used as a reference to the result of an asynchronous computation. It provides an isDone() method to check whether the computation is done or not, and a get() method to retrieve the result of the computation when it is done.