Reentrant Lock in Java

Background

The traditional way to achieve thread synchronization in Java is by the use of synchronized keyword. While it provides a certain basic synchronization, the synchronized keyword is quite rigid in its use. For example, a thread can take a lock only once. Synchronized blocks don't offer any mechanism of a waiting queue and after the exit of one thread, any thread can take the lock. This could lead to starvation of resources for some other thread for a very long period of time.

Reentrant Locks are provided in Java to provide synchronization with greater flexibility.

What are Reentrant Locks?

The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.

As the name says, ReentrantLock allow threads to enter into lock on a resource more than once. When the thread first enters into lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

Reentrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request i.e. after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.

These locks are used in the following way:

```
public void some_method()
{
        reentrantlock.lock();
        try
        {
            //Do some work
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        finally
        {
            reentrantlock.unlock();
        }

}
```

The unlock statement is always called in the finally block to ensure that the lock is released even if an exception is thrown in the method body(try block).

ReentrantLock Methods

- **lock():** Call to the lock() method increments the hold count by 1 and gives the lock to the thread if the shared resource is initially free.
- **unlock():** Call to the unlock() method decrements the hold count by 1. When this count reaches zero, the resource is released.
- **tryLock():** If the resource is not held by any other thread, then call to tryLock() returns true and the hold count is incremented by one. If the resource is not free then the method returns false and the thread is not blocked but it exits.
- **tryLock(long timeout, TimeUnit unit):** As per the method, the thread waits for a certain time period as defined by arguments of the method to acquire the lock on the resource before exiting.
- **lockInterruptibly():** This method acquires the lock if the resource is free while allowing for the thread to be interrupted by some other thread while acquiring the resource. It means that if the current thread is waiting for lock but some other thread requests the lock, then the current thread will be interrupted and return immediately without acquiring lock.
- **getHoldCount():** This method returns the count of the number of locks held on the resource.
- **isHeldByCurrentThread():** This method returns true if the lock on the resource is held by the current thread.

ReentrantLock Example

In the following tutorial, we will look at a basic example of Reentrant Locks.

Steps to be followed

```
1. Create an object of ReentrantLock
2. Create a worker(Runnable Object) to execute and pass the lock to the object
3. Use the lock() method to acquire the lock on shared resource
4. After completing work, call unlock() method to release the lock
```