# ReentrantLock

This lesson explains usage of the ReentrantLock.

**We'll cover the following**



*If you are interviewing, consider buying our number#1 course for Java Multithreading Interviews (https://bit.ly/2QfKXCK).*

The `ReentrantLock` implements the `Lock` interface and is functionally similar to the implicit monitor lock accessed using `synchronized` methods and statements.

The lock is said to be owned by the thread that `lock`s it and any other thread attempting to `lock` the object will block. A thread that already owns the lock will return immediately if it invokes `lock` again. The reentrant behavior of the lock allows recursively locking by the already owning thread, however, the lock supports a maximum of 2147483647 locks by the same thread.

# Idiomatic Use of Lock#

Threads can experience deadlocks when locks aren't unlocked after use. The correct idiomatic usage of a lock should follow the below pattern:

```
lock.lock();  // acquire the lock
     try {
       // … functionality to be executed in the method body
     } finally {
       lock.unlock() // must release the lock in a finally block
     }
```

If you acquire a lock and then continue execution without a `try` block it is possible that an exception occurs and the lock is never released even though the program continues to execute. Depending on how the program is structured it is possible that the program experiences a deadline as it progresses.

# Fairness#

The `ReentrantLock` can also be operated in *fair mode* where the lock is granted to the longest waiting thread. Thus no thread experiences starvation and the variance in times to obtain the lock is also small. Without the *fair mode* the lock doesn't guarantee the order in which threads acquire the lock. When a lock is operated in fair mode in an environment with several threads contending access to the lock, throughput suffers and is significantly reduced. Finally, the method `tryLock` when invoked without timeout doesn't honor the fairness setting and acquires the lock if it is free even in the presence of other waiting threads.

# Example#

Consider the example below which has three threads, the main thread, threadA and threadB interacting with an instance of `ReentrantLock`. The main thread locks the lock object thrice and invokes `unlock` on the object with an artificially introduced delay. During this time, `threadA` does a busy spinning waiting for the lock to be free. `threadA` uses the method

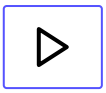
spinning waiting for the lock to be free. `threadA` uses the method `tryLock()` to check if it can acquire the lock. On the other hand `threadB` simply acquires and then releases the lock. `threadB` will inevitably block at acquiring the lock and the method `getQueueLength()` will display a count of 1. The example also demonstrates the use of the method `getHoldCount()` which either returns 0 if the lock isn't held by the thread invoking the method or the number of times the owning thread has recursively acquired the lock. Pay attention to how we use this method in the `finally` block of the main thread to `unlock` as many times as required if an exception occurs.

Finally, there's the `isLocked()` method that returns true if the lock is held by any thread. However, note that both methods `isLocked()` and `getQueueLength()` are designed for monitoring the state of the system and shouldn't be used in program control, e.g. you should never do something like below:

```
        if (lock.isLocked()) {
            // Take some action
        }
```

```
 1  import java.util.concurrent.ExecutorService;
 2  import java.util.concurrent.Executors;
 3  import java.util.concurrent.Future;
 4  import java.util.concurrent.locks.Lock;
 5  import java.util.concurrent.locks.ReentrantLock;
 6
 7  class Demonstration {
 8      public static void main( String args[] ) throws Exception {
 9
10          ExecutorService es = Executors.newFixedThreadPool(5);
11          ReentrantLock lock = new ReentrantLock();
12          Runnable threadA = new Runnable() {
13              @Override
14              public void run() {
15                  threadA(lock);
16              }
17          };
18
19          Runnable threadB = new Runnable() {
```

```
20              @Override
21              public void run() {
22                  threadB(lock);
23              }
24          };
25
26          try {
27              lock.lock();
28              lock.lock();
```




Q Consider the program below

```
public class question {

    static ReentrantLock lock = new ReentrantLock();

    static int test() {

        lock.lock();

        try {
            return -1;
        } finally {
            lock.unlock();
            System.out.println("Unlocked");
        }

        System.out.println("Exiting Program");
    }
}
```

A) “Unlocked” will be printed but “Exiting Program” is unreachable statement.

B) Both print statements are unreachable.

C) Both print statements are executed.

D) `lock` object is never unlocked because of the `return` statement.

Submit Answer

Reset Quiz



Back

Lock Interface

Next

ReadWriteLock

Mark as Completed