Java ReentrantLock

(Part 2)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand how the concept of mutual exclusion in concurrent programs
- Recognize how Java ReentrantLock provides mutual exclusion to concurrent programs
Overview of ReentrantLock
Overview of ReentrantLock

• Provide mutual exclusion to concurrent Java programs

public class ReentrantLock
  implements Lock,
  java.io.Serializable {

Class ReentrantLock

java.lang.Object
  java.util.concurrent.locks.ReentrantLock

All Implemented Interfaces:
  Serializable, Lock

public class ReentrantLock
extends Object
implements Lock, Serializable

A reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.

A ReentrantLock is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock. This can be checked using methods isHeldByCurrentThread(), and getHoldCount().

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html
**Overview of ReentrantLock**

- Provide mutual exclusion to concurrent Java programs
- Implements Lock interface

```java
public class ReentrantLock implements Lock, java.io.Serializable {
...
```

### Interface Lock

**All Known Implementing Classes:**
- ReentrantLock, ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock

```java
public interface Lock
```

**Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements. They allow more flexible structuring, may have quite different properties, and may support multiple associated Condition objects.**

A lock is a tool for controlling access to a shared resource by multiple threads. Commonly, a lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and all access to the shared resource requires that the lock be acquired first. However, some locks may allow concurrent access to a shared resource, such as the read lock of a ReentrantReadWriteLock.

The use of synchronized methods or statements provides access to the implicit monitor lock associated with every object, but forces all lock acquisition and release to occur in a block-structured way: when multiple locks are acquired they must be released in the opposite order, and all locks must be released in the same lexical scope in which they were acquired.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html)
Overview of ReentrantLock

- Applies the *Bridge* pattern

```
public class ReentrantLock
    implements Lock, java.io.Serializable {
    ...

Decouples its interface from its implementation so fair & non-fair semantics can be supported uniformly
```

See [en.wikipedia.org/wiki/Bridge_pattern](en.wikipedia.org/wiki/Bridge_pattern)
Overview of ReentrantLock

• Applies the *Bridge* pattern
• Locking handled by Sync Implementor hierarchy

```java
public class ReentrantLock
    implements Lock,
    java.io.Serializable {

    ... 

    /** Performs sync mechanics */
    final Sync sync;
```
Overview of ReentrantLock

• Applies the *Bridge* pattern
• Locking handled by Sync Implementor hierarchy
• Inherits functionality from AbstractQueuedSynchronizer

```java
public class ReentrantLock implements Lock, java.io.Serializable {

    /* Performs sync mechanics */
    final Sync sync;

    /* Sync implementation for ReentrantLock */
    abstract static class Sync extends AbstractQueuedSynchronizer {
        ...
    }

    ...
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html)
Overview of ReentrantLock

- Applies the *Bridge* pattern
- Locking handled by Sync Implementor hierarchy
- Inherits functionality from AbstractQueuedSynchronizer
- Many Java synchronizers that use FIFO wait queues use this framework

```java
public class ReentrantLock implements Lock, java.io.Serializable {
  ...
  /** Performs sync mechanics */
  final Sync sync;
  
  /** Sync implementation for ReentrantLock */
  abstract static class Sync extends AbstractQueuedSynchronizer {
    { ... }
  }
  ...
}
```

See [gee.cs.oswego.edu/dl/papers/aqs.pdf](gee.cs.oswego.edu/dl/papers/aqs.pdf)
Overview of ReentrantLock

- Applies the Bridge pattern
- Locking handled by Sync Implementor hierarchy
- Inherits functionality from AbstractQueuedSynchronizer
- Defines NonFairSync & FairSync subclasses with non-FIFO & FIFO semantics

public class ReentrantLock implements Lock, java.io.Serializable {
    ...
    /** Performs sync mechanics */
    final Sync sync;

    /** Sync implementation for ReentrantLock */
    abstract static class Sync extends AbstractQueuedSynchronizer {
        ...
    }

    static final class NonFairSync extends Sync {
    }

    static final class FairSync extends Sync {
    }

See src/share/classes/java/util/concurrent/locks/ReentrantLock.java
Overview of ReentrantLock

- Applies the *Bridge* pattern
- Locking handled by Sync Implementor hierarchy
- Inherits functionality from AbstractQueuedSynchronizer
- Defines NonFairSync & FairSync subclasses with non-FIFO & FIFO semantics
- Constructor enables use to select fair or non-fair lock acquisition model

```java
public class ReentrantLock
    implements Lock, java.io.Serializable {
    ...
    public ReentrantLock
        (boolean fair) {
        sync = fair
            ? new FairSync()
            : new NonfairSync();
    }
    ...
    
    This param determines whether FairSync or NonfairSync is used
```

The Reentrantlock fair & non-fair models follow the same pattern used by the Java Semaphore
Overview of ReentrantLock

• Applies the Bridge pattern
• Locking handled by Sync Implementor hierarchy
  • Inherits functionality from AbstractQueuedSynchronizer
• Defines NonFairSync & FairSync subclasses with non-FIFO & FIFO semantics
• Constructor enables use to select fair or non-fair lock acquisition model

```java
public class ReentrantLock implements Lock, java.io.Serializable {

  public ReentrantLock (boolean fair) {
    sync = fair
      ? new FairSync()
      : new NonfairSync();
  }

  ... Ensures strict “FIFO” fairness, at the expense of performance

...}
```
Overview of ReentrantLock

• Applies the *Bridge* pattern
• Locking handled by Sync Implementor hierarchy
  • Inherits functionality from AbstractQueuedSynchronizer
• Defines NonFairSync & FairSync subclasses with non-FIFO & FIFO semantics
• Constructor enables use to select fair or non-fair lock acquisition model

```java
public class ReentrantLock
    implements Lock, java.io.Serializable {

    public ReentrantLock (boolean fair) {
        sync = fair
            ? new FairSync()
            : new NonfairSync();
    }

    ... Enables faster performance at the expense of fairness ...
```
Overview of ReentrantLock

- Applies the *Bridge* pattern
- Locking handled by Sync Implementor hierarchy
- Inherits functionality from AbstractQueuedSynchronizer
- Defines NonFairSync & FairSync subclasses with non-FIFO & FIFO semantics
- Constructor enables use to select fair or non-fair lock acquisition model

```java
public class ReentrantLock
    implements Lock,
    java.io.Serializable {
    ...

    public ReentrantLock
        (boolean fair) {
        sync = fair
            ? new FairSync()
            : new NonfairSync();
    }

    public ReentrantLock() {
        sync = new NonfairSync();
    }
    ...

    The default behavior favors performance over fairness
```
Overview of ReentrantLock

• Applies the *Bridge* pattern
• Locking handled by Sync Implementor hierarchy
• Inherits functionality from AbstractQueuedSynchronizer
• Defines NonFairSync & FairSync subclasses with non-FIFO & FIFO semantics
• Constructor enables use to select fair or non-fair lock acquisition model

```java
public class ReentrantLock
    implements Lock,
    java.io.Serializable {

    ...  

    public ReentrantLock
        (boolean fair) {
        sync = fair
            ? new FairSync()
            : new NonfairSync();
    }

    public ReentrantLock() {
        sync = new NonfairSync();
    }

    ...  

    FairSync is generally much slower than NonfairSync, so use it accordingly

```
### Overview of ReentrantLock

- ReentrantLock is similar to the monitor lock provided by Java’s built-in monitor objects

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void lock()</td>
<td>Acquires the lock</td>
</tr>
<tr>
<td>void unlock()</td>
<td>Attempts to release this lock</td>
</tr>
</tbody>
</table>

- void tryLock() – Acquires the lock only if it is not held by another thread at the time of invocation
- void tryLock(long timeout, TimeUnit unit) – Acquires the lock if it is not held by another thread within the given waiting time and the current thread has not been interrupted

See upcoming lessons on “Java Built-in Monitor Object”
Overview of ReentrantLock

- ReentrantLock is similar to the monitor lock provided by Java’s built-in monitor objects
- But also provides extended capabilities

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void lock()</td>
<td>Acquires the lock</td>
</tr>
<tr>
<td>void unlock()</td>
<td>Attempts to release this lock</td>
</tr>
<tr>
<td>void lockInterruptibly()</td>
<td>Acquires the lock unless the current thread is interrupted</td>
</tr>
<tr>
<td>boolean tryLock()</td>
<td>Acquires the lock only if it is not held by another thread at the time of invocation</td>
</tr>
<tr>
<td>boolean tryLock(long timeout, TimeUnit unit)</td>
<td>Acquires the lock if it is not held by another thread within the given waiting time and the current thread has not been interrupted</td>
</tr>
</tbody>
</table>
Overview of ReentrantLock

• ReentrantLock is similar to the monitor lock provided by Java’s built-in monitor objects
• But also provides extended capabilities

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void lock()</td>
<td>Acquires the lock</td>
</tr>
<tr>
<td>void unlock()</td>
<td>Attempts to release this lock</td>
</tr>
<tr>
<td>void lockInterruptibly()</td>
<td>Acquires the lock unless the current thread is interrupted</td>
</tr>
<tr>
<td>boolean tryLock()</td>
<td>Acquires the lock only if it is not held by another thread at the time of invocation</td>
</tr>
<tr>
<td>boolean tryLock(long timeout, TimeUnit unit)</td>
<td>Acquires the lock if it is not held by another thread within the given waiting time and the current thread has not been interrupted</td>
</tr>
</tbody>
</table>

In contrast, Java’s synchronized methods/statements are not interruptible
Overview of ReentrantLock

• ReentrantLock is similar to the monitor lock provided by Java’s built-in monitor objects
  • But also provides extended capabilities

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void lock()</td>
<td>Acquires the lock</td>
</tr>
<tr>
<td>void unlock()</td>
<td>Attempts to release this lock</td>
</tr>
<tr>
<td>void lockInterruptibly()</td>
<td>Acquires the lock unless the current thread is interrupted</td>
</tr>
<tr>
<td>boolean tryLock()</td>
<td>Acquires the lock only if it is not held by another thread at the time of invocation</td>
</tr>
<tr>
<td>boolean tryLock(long timeout, TimeUnit unit)</td>
<td>Acquires the lock if it is not held by another thread within the given waiting time and the current thread has not been interrupted</td>
</tr>
</tbody>
</table>

Likewise, Java’s synchronized methods/statements aren’t non-blocking
Overview of ReentrantLock

- A ReentrantLock supports “recursive mutex” semantics

See en.wikipedia.org/wiki/Reentrant_mutex
Overview of ReentrantLock

- A ReentrantLock supports “recursive mutex” semantics
- The thread that hold the mutex can reacquire it without self-deadlock
Overview of ReentrantLock

- A ReentrantLock supports “recursive mutex” semantics
- The thread that hold the mutex can reacquire it without self-deadlock
Recursive mutex semantics add a bit more overhead relative to non-recursive semantics due to extra software logic & synchronization.

```java
boolean nonfairTryAcquire(int acquires) {
    Thread t =
        Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(t);
            return true;
        }
    } else if (t == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        ...
        setState(nextc);
        return true;
    }
    return false;
}
```

See [src/share/classes/java/util/concurrent/locks/ReentrantLock.java](src/share/classes/java/util/concurrent/locks/ReentrantLock.java)
Overview of ReentrantLock

- Recursive mutex semantics add a bit more overhead relative to non-recursive semantics due to extra software logic & synchronization.

```java
boolean nonfairTryAcquire(int acquires) {
    Thread t = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(t);
            return true;
        }
    } else if (t == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        ...
        setState(nextc);
        return true;
    }
    return false;
}
```

Atomically acquire the lock if it's available.

See src/share/classes/java/util/concurrent/locks/ReentrantLock.java
Overview of ReentrantLock

- Recursive mutex semantics add a bit more overhead relative to non-recursive semantics due to extra software logic & synchronization.

```java
boolean nonfairTryAcquire
    (int acquire)
{
    Thread t = 
        Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0,
            acquire)) {
            setExclusiveOwnerThread(t);
            return true;
        }
    } else if (t ==
        getExclusiveOwnerThread()) {
        int nextc = c + acquire;
        ...
        setState(nextc);
        return true;
    }
    return false;
}
```

Simply increment lock count if the current thread is lock owner.

See `src/share/classes/java/util/concurrent/locks/ReentrantLock.java`
Overview of ReentrantLock

- ReentrantLock semantics are useful for frameworks that hold locks during callbacks to user code

```java
mLock.lock();
try {
    mCancelled = true;
    mSchedExeSvc.shutdownNow();
} finally {
    mLock.unlock();
}
```

```java
mLock.lock();
try {
    if (...) 
        cancel();
    ... 
} finally {
    mLock.unlock();
}
```

Overview of ReentrantLock

- ReentrantLock semantics are useful for frameworks that hold locks during callbacks to user code.

```java
mLock.lock();
try {
    mCancelled = true;
    mSchedExeSvc.shutdownNow();
} finally {
    mLock.unlock();
}
```

- `mLock.lock();` is called before any callback methods are invoked.
- `mLock.unlock();` is called after all callback methods are finished to ensure the lock is released.

```
if (...) {
    cancel();
}
```

- `cancel()` method is called to immediately cancel the timer.

Framework calls `onTick()` hook method with the `mLock` held.

```java
mLock.lock();
try {
    ...
    onTick(millisLeft);
    ...
} finally {
    mLock.unlock();
}
```
Overview of ReentrantLock

- ReentrantLock semantics are useful for frameworks that hold locks during callbacks to user code

```java
mLock.lock();
try {
    mCancelled = true;
    mSchedExeSvc.shutdownNow();
} finally {
    mLock.unlock();
}
```

The app-defined onTick() hook method can call cancel

```java
mLock.lock();
try {
    if (...) 
        cancel();
    ... 
    onTick(millisLeft);
} finally {
    mLock.unlock();
}
```

• ReentrantLock semantics are useful for frameworks that hold locks during callbacks to user code

```java
mLock.lock();
try {
    mCancelled = true;
    mSchedExeSvc.shutdownNow();
} finally {
    mLock.unlock();
}
```

```java
if (...) cancel();
```

`cancel()` also acquires `mLock`, which must be recursive or self-deadlock will occur.

End of Java
ReentrantLock (Part 2)