Java ReentrantLock
(Part 1)

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

See en.wikipedia.org/wiki/Mutual_exclusion
Overview of Mutual Exclusion Locks
Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a “critical section”

See en.wikipedia.org/wiki/Critical_section
Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a "critical section"
- Ensures only one thread can run in a block of code at a time
Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a “critical section”
- Ensures only one thread can run in a block of code at a time
Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a “critical section”
- Ensures only one thread can run in a block of code at a time

![Diagram of critical section and running thread]
Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a "critical section"
  - Ensures only one thread can run in a block of code at a time
- Other threads are kept "at bay" so they don’t corrupt shared resources that can be set/get by multiple concurrent operations
A mutual exclusion lock (mutex) defines a “critical section”

- Ensures only one thread can run in a block of code at a time

- Other threads are kept “at bay” so they don’t corrupt shared resources that can be set/get by multiple concurrent operations

- Race conditions could occur if multiple threads could run within a critical section

Race conditions can arise when a program depends on the sequence or timing of threads for it to operate properly

Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a “critical section”
  - Ensures only one thread can run in a block of code at a time
  - Other threads are kept “at bay” so they don’t corrupt shared resources that can be set/get by multiple concurrent operations
  - After a thread leaves a critical section another thread can enter & start running
Overview of Mutual Exclusion Locks

• A mutual exclusion lock (mutex) defines a “critical section”
  • Ensures only one thread can run in a block of code at a time
  • Other threads are kept “at bay” so they don’t corrupt shared resources that can be set/get by multiple concurrent operations
  • After a thread leaves a critical section another thread can enter & start running
Overview of Mutual Exclusion Locks

- A mutual exclusion lock (mutex) defines a “critical section”
  - Ensures only one thread can run in a block of code at a time
  - Other threads are kept “at bay” so they don’t corrupt shared resources that can be set/get by multiple concurrent operations
  - After a thread leaves a critical section another thread can enter & start running
Overview of Mutual Exclusion Locks

• A mutex is typically implemented in hardware via atomic operations

Atomic operations appear to occur instantaneously & either change the state of the system successful or have no effect.

See en.wikipedia.org/wiki/Linearizability
Overview of Mutual Exclusion Locks

- A mutex is typically implemented in hardware via atomic operations
- Implemented in Java via the compareAndSwap*() methods in the Unsafe class

Concurrency

And few words about concurrency with Unsafe. compareAndSwap methods are atomic and can be used to implement high-performance lock-free data structures.

For example, consider the problem to increment value in the shared object using lot of threads.

First we define simple interface Counter:

```java
interface Counter {
    void increment();
    long getCount();
}
```

Then we define worker thread CounterClient, that uses Counter:

```java
class CounterClient implements Runnable {
    private Counter c;
    private int num;

    public CounterClient(Counter c, int num) {
        this.c = c;
        this.num = num;
    }

    @Override
    public void run() {
        for (int i = 0; i < num; i++) {
            c.increment();
        }
    }
}
```

See earlier discussion of "Java Atomic Classes & Operations"
Human Known Use of Mutual Exclusion Locks
Human Known Use of Mutual Exclusion Locks

- A human known use of mutual exclusion locks is an airplane restroom protocol
Human Known Use of Mutual Exclusion Locks

- A human known use of mutual exclusion locks is an airplane restroom protocol.
Human Known Use of Mutual Exclusion Locks

• A human known use of mutual exclusion locks is an airplane restroom protocol.
Human Known Use of Mutual Exclusion Locks

- A human known use of mutual exclusion locks is an airplane restroom protocol.
A human known use of mutual exclusion locks is an airplane restroom protocol.

This protocol is “fully-bracketed,” i.e., person who locks must be the same as the person who unlocks.
End of Java
ReentrantLock (Part 1)
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

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

### Class ReentrantLock

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

**All Implemented Interfaces:**
- Serializable, Lock

```java
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](http://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

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

Interface Lock

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

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
• Applies the *Bridge* pattern

```java
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](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 rely on 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
- Optionally implements 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
- Optionally implements fair or non-fair lock acquisition model

```
public class ReentrantLock
    implements Lock, java.io.Serializable {
    ...
    public ReentrantLock
        (boolean fair) {
        sync = fair
            ? new FairSync()
            : new NonfairSync();
    }
    ...
```
Overview of ReentrantLock

- Applies the Bridge pattern
- Locking handled by Sync Implementor hierarchy
- Inherits functionality from AbstractQueuedSynchronizer
- Optionally implements 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
- Optionally implements 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
- Optionally implements 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 Name</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>

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>void</th>
<th>lock() – Acquires the lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>unlock() – Attempts to release this lock</td>
</tr>
<tr>
<td>void</td>
<td>lockInterruptibly() – Acquires the lock unless the current thread is interrupted</td>
</tr>
<tr>
<td>boolean</td>
<td>tryLock() – Acquires the lock only if it is not held by another thread at the time of invocation</td>
</tr>
<tr>
<td>boolean</td>
<td>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</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>

Nor are Java’s synchronized methods/statements 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
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 additional software logic & synchronization
Overview of Key ReentrantLock Methods
Overview of Key ReentrantLock Methods

- It key methods acquire & release the lock

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

    ...

    public void lock() { sync.lock(); }

    public void lockInterruptibly()
        throws InterruptedException {
        sync.acquireInterruptibly(1);
    }

    public boolean tryLock() {
        return sync.nonfairTryAcquire(1);
    }

    public void unlock() {
        sync.release(1);
    }

    ...
```
Overview of Key ReentrantLock Methods

- It key methods acquire & release the lock

```java
public class ReentrantLock
    implements Lock,
    java.io.Serializable {
    ...
    public void lock() { sync.lock(); }

    public void lockInterruptibly()
        throws InterruptedException {
            sync.acquireInterruptibly(1);
    }

    public boolean tryLock() {
        return sync.nonfairTryAcquire(1);
    }

    public void unlock() {
        sync.release(1);
    }
    ...
```

These methods are defined in the Java Lock interface
Overview of Key ReentrantLock Methods

- It key methods acquire & release the lock

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

    ...

    public void lock() { sync.lock(); }

    public void lockInterruptibly()
        throws InterruptedException {
            sync.acquireInterruptibly(1);
    }

    public boolean tryLock() {
        return sync.nonfairTryAcquire(1);
    }

    public void unlock() {
        sync.release(1);
    }

    ...

    These methods all simply forward to their implementor methods, most of which are inherited from the AbstractQueuedSynchronizer framework
```
Overview of Key ReentrantLock Methods

- It key methods acquire & release the lock
- `lock()` acquires the lock if it’s available

```java
class ReentrantLock implements Lock, java.io.Serializable {
    ...
    public void lock() {
        sync.lock();
    }
    ...
}
```
Overview of Key ReentrantLock Methods

- It key methods acquire & release the lock
- `lock()` acquires the lock if it’s available
- If lock isn’t available its implementation depends on the “fairness” policy

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

    ...

    public void lock() {
        sync.lock();
    }

    ...

    ...
```
Overview of Key ReentrantLock Methods

- It key methods acquire & release the lock
- lock() acquires the lock if it’s available
- If lock isn’t available its implementation depends on the “fairness” policy
- Non-fair implementations are optimized in hardware

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

    public void lock() {
        sync.lock();
    }

    ...

See en.wikipedia.org/wiki/Spinlock
Overview of Key ReentrantLock Methods

- It key methods acquire & release the lock
  - lock() acquires the lock if it’s available
  - If lock isn’t available its implementation depends on the “fairness” policy
    - Non-fair implementations are optimized in hardware
    - Fair implementations “park” themselves on a wait queue

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

    public void lock() {
        sync.lock();
    }

    ...
```
Overview of Key ReentrantLock Methods

• It key methods acquire & release the lock
  • lock() acquires the lock if it’s available
  • lockInterruptibly() acquires lock unless interrupted

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

    ...  
    public void lockInterruptibly() 
        throws InterruptedException {
        sync.acquireInterruptibly(1);
    }

    ...  
```

See lesson on “Managing the Java Thread Lifecycle”
Overview of Key ReentrantLock Methods

- It key methods acquire & release the lock
  - `lock()` acquires the lock if it’s available
  - `lockInterruptibly()` acquires lock unless interrupted
  - In contrast, `lock()` is not interruptible

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

    ...
    public void lockInterruptibly()
        throws InterruptedException {
        sync.acquireInterruptibly(1);
    }

    ...
    ...
```
Overview of Key ReentrantLock Methods

- It key methods acquire & release the lock
- lock() acquires the lock if it’s available
- lockInterruptibly() acquires lock unless interrupted
- tryLock() acquires lock only if it’s not held by another thread at invocation time

```java
class ReentrantLock implements Lock, java.io.Serializable {
    public boolean tryLock() {
        sync.nonfairTryAcquire(1);
    }
    ...
}
```

Untimed tryLock() doesn’t honor fairness setting & can “barge”
Overview of Key ReentrantLock Methods

• It key methods acquire & release the lock
• lock() acquires the lock if it’s available
• lockInterruptibly() acquires lock unless interrupted
• tryLock() acquires lock only if it’s not held by another thread at invocation time
• unlock() attempts to release the lock

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

    ...      
    public void unlock() {
        sync.release(1);
    }
    ...
Overview of Key ReentrantLock Methods

- It key methods acquire & release the lock
  - lock() acquires the lock if it’s available
  - lockInterruptibly() acquires lock unless interrupted
  - tryLock() acquires lock only if it’s not held by another thread at invocation time
  - unlock() attempts to release the lock

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

    ...
    public void unlock() {
        sync.release(1);
    }

    ...
```
Overview of Key ReentrantLock Methods

- It key methods acquire & release the lock
  - lock() acquires the lock if it’s available
  - lockInterruptibly() acquires lock unless interrupted
  - tryLock() acquires lock only if it’s not held by another thread at invocation time
  - unlock() attempts to release the lock
    - IllegalMonitorStateException is thrown if calling thread doesn’t hold lock

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

    public void unlock() {
        sync.release(1);
    }

    ...
}
```

i.e., a ReentrantLock is “fully bracketed”!
Overview of Other ReentrantLock Methods
### Overview of Other ReentrantLock Methods

- There are many other ReentrantLock methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean <code>tryLock()</code></td>
<td>(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</td>
</tr>
<tr>
<td>boolean <code>isFair()</code></td>
<td>– Returns true if this lock has fairness set true</td>
</tr>
<tr>
<td>boolean <code>isLocked()</code></td>
<td>– Queries if this lock is held by any thread</td>
</tr>
<tr>
<td>Condition <code>newCondition()</code></td>
<td>– Returns a Condition instance for use with this Lock instance</td>
</tr>
</tbody>
</table>
Overview of Other ReentrantLock Methods

- There are many other ReentrantLock methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boolean tryLock()</code></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>
<tr>
<td><code>boolean isFair()</code></td>
<td>Returns true if this lock has fairness set true.</td>
</tr>
<tr>
<td><code>boolean isLocked()</code></td>
<td>Queries if this lock is held by any thread.</td>
</tr>
<tr>
<td><code>Condition newCondition()</code></td>
<td>Returns a Condition instance for use with this Lock instance.</td>
</tr>
</tbody>
</table>

Timed tryLock() does honor fairness setting & can’t “barge”
Overview of Other ReentrantLock Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>boolean isFair()</td>
<td>Returns true if this lock has fairness set true</td>
</tr>
<tr>
<td>boolean isLocked()</td>
<td>Queries if this lock is held by any thread</td>
</tr>
<tr>
<td>Condition newCondition()</td>
<td>Returns a Condition instance for use with this Lock instance</td>
</tr>
</tbody>
</table>

...
Overview of Other ReentrantLock Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>boolean isFair()</td>
<td>Returns true if this lock has fairness set true</td>
</tr>
<tr>
<td>boolean isLocked()</td>
<td>Queries if this lock is held by any thread</td>
</tr>
<tr>
<td>Condition newCondition()</td>
<td>Returns a Condition instance for use with this Lock instance</td>
</tr>
</tbody>
</table>

... ... ...

Not very useful due to non-determinism of concurrency..
There are many other ReentrantLock methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>boolean isFair()</td>
<td>Returns true if this lock has fairness set true</td>
</tr>
<tr>
<td>boolean isLocked()</td>
<td>Queries if this lock is held by any thread</td>
</tr>
<tr>
<td>Condition newCondition()</td>
<td>Returns a Condition instance for use with this Lock instance</td>
</tr>
</tbody>
</table>

See upcoming lesson on "Java ConditionObject"
End of Java ReentrantLock (Part 2)