Java ReentrantLock: Structure & Functionality

Douglas C. Schmidt

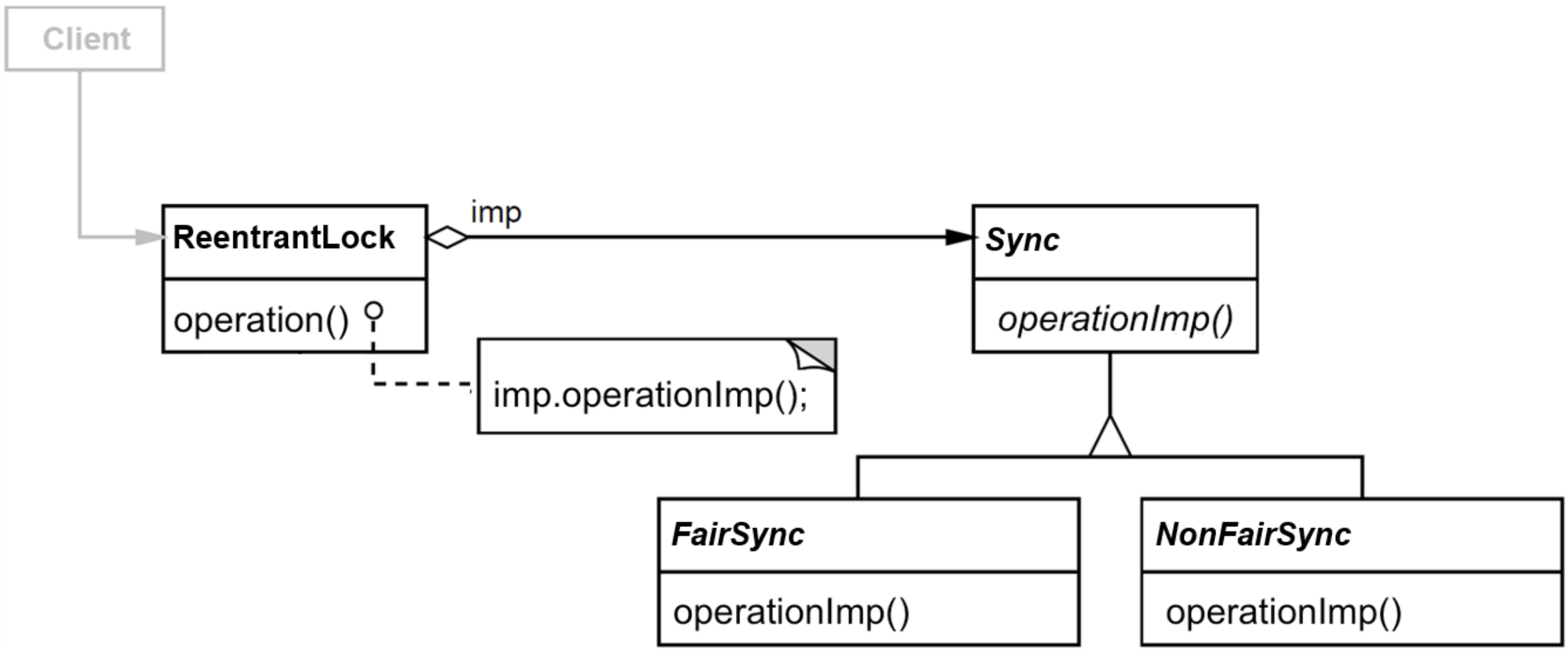
d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

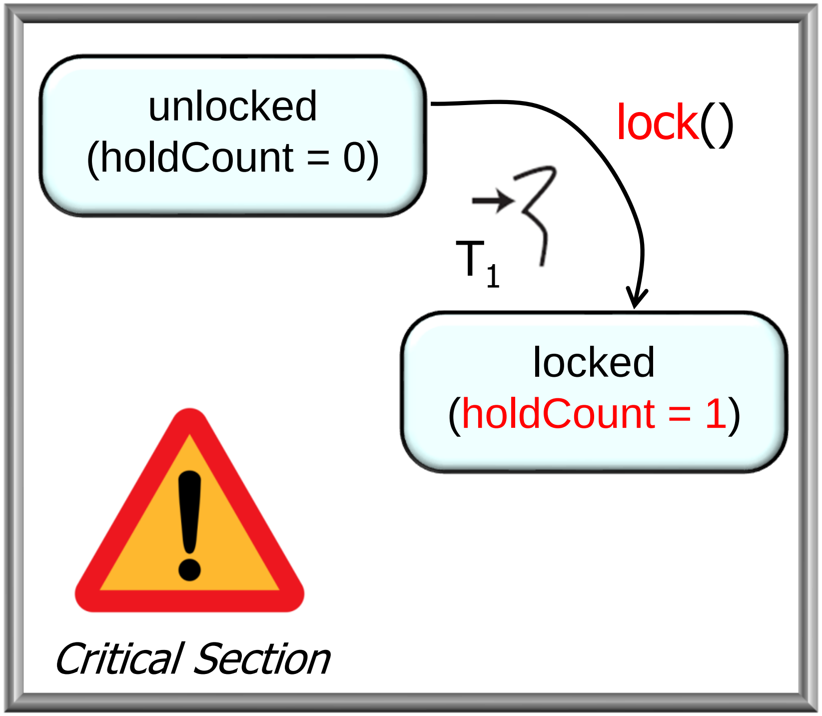
Learning Objectives in this Part of the Lesson

- Understand the concept of mutual exclusion in concurrent programs
- Note a human-known use of mutual exclusion
- Recognize the structure & functionality of Java ReentrantLock



Learning Objectives in this Part of the Lesson

- Understand the concept of mutual exclusion in concurrent programs
- Note a human-known use of mutual exclusion
- Recognize the structure & functionality of Java ReentrantLock
- Be aware of reentrant mutex semantics



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

```
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 `ReadWriteLock`.

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.

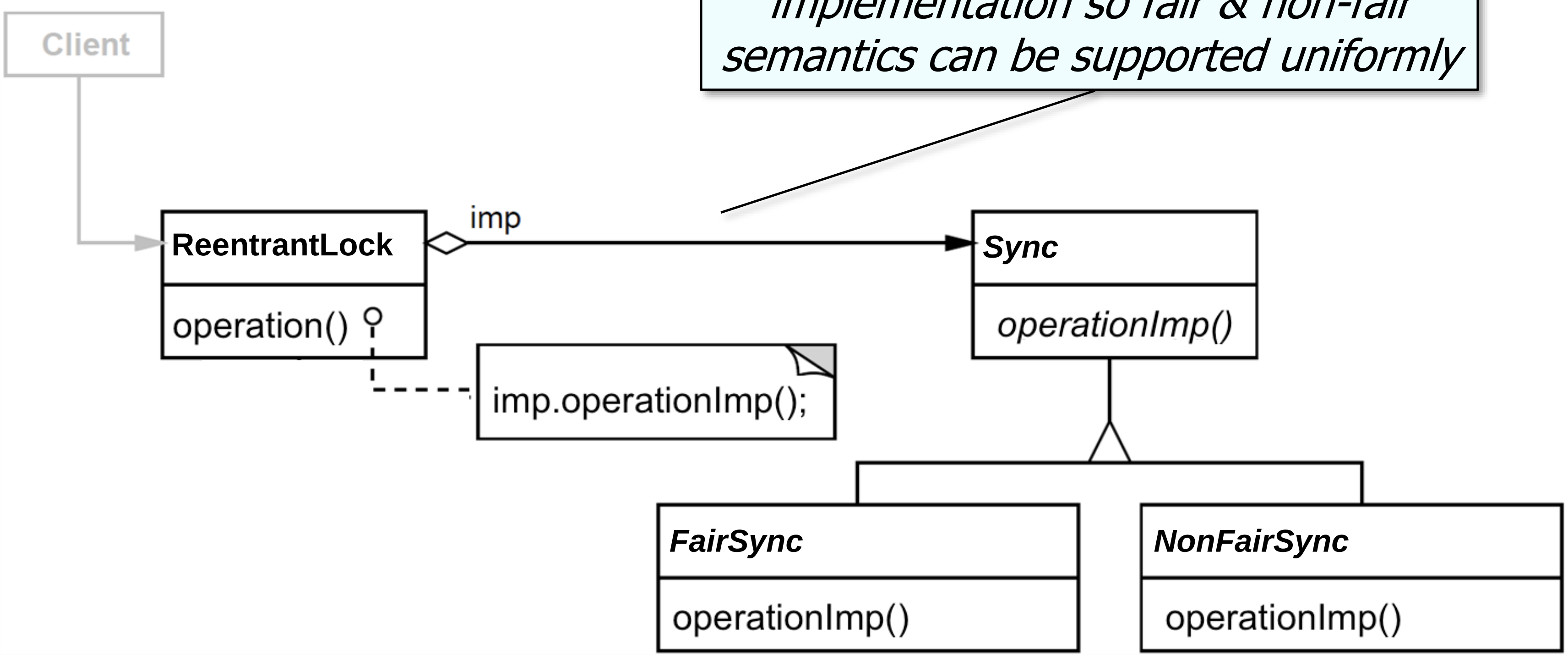
See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html

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

Overview of ReentrantLock

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

```
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

```
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

Overview of ReentrantLock

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

```
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

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
  - Constructor enables fair vs. non-fair lock acquisition model

```
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

Overview of ReentrantLock

- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Constructor enables fair vs. non-fair lock acquisition model
    - These models apply the same pattern used by Semaphore & ReentrantReadWriteLock

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

See upcoming lessons on "*Java Semaphore*" & "*Java ReentrantReadWriteLock*"

Overview of ReentrantLock

- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Constructor enables fair vs. non-fair lock acquisition model
    - These models apply the same pattern used by Semaphore & ReentrantReadWriteLock

```
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
  - Constructor enables fair vs. non-fair lock acquisition model
    - These models apply the same pattern used by Semaphore & ReentrantReadWriteLock

```
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
  - Constructor enables fair vs. non-fair lock acquisition model
    - These models apply the same pattern used by Semaphore & ReentrantReadWriteLock

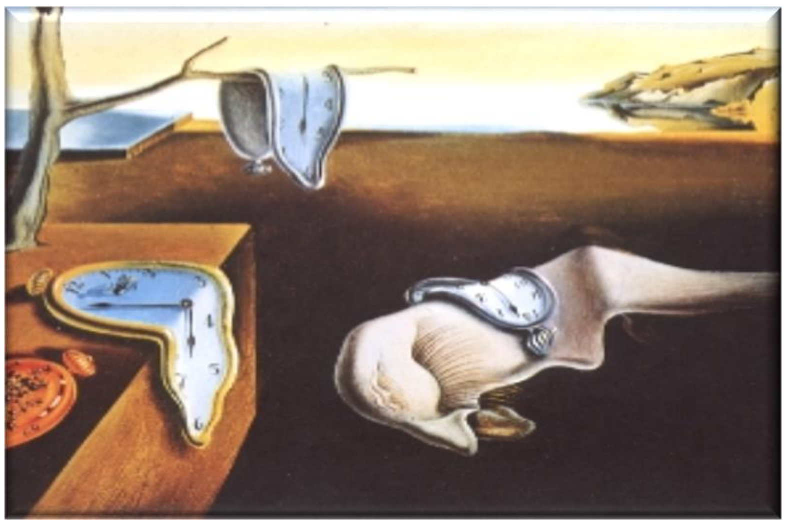
```
public class ReentrantLock
             implements Lock,
             java.io.Serializable {
  ...
  public ReentrantLock
           (boolean fair) {
    sync = fair
      ? new FairSync()
      : new NonfairSync();
  }

  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
- Constructor enables fair vs. non-fair lock acquisition model
  - These models apply the same pattern used by Semaphore & ReentrantReadWriteLock



```
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

void	lock() – Acquires the lock
void	unlock() – Attempts to release this lock

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

void	lock() – Acquires the lock
void	unlock() – Attempts to release this lock
void	lockInterruptibly() – Acquires the lock unless the current thread is interrupted
boolean	tryLock() – Acquires the lock only if it is not held by another thread at the time of invocation
boolean	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

Overview of ReentrantLock

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

void	lock() – Acquires the lock
void	unlock() – Attempts to release this lock
void	lockInterruptibly() – Acquires the lock unless the current thread is interrupted
boolean	tryLock() – Acquires the lock only if it is not held by another thread at the time of invocation
boolean	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

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

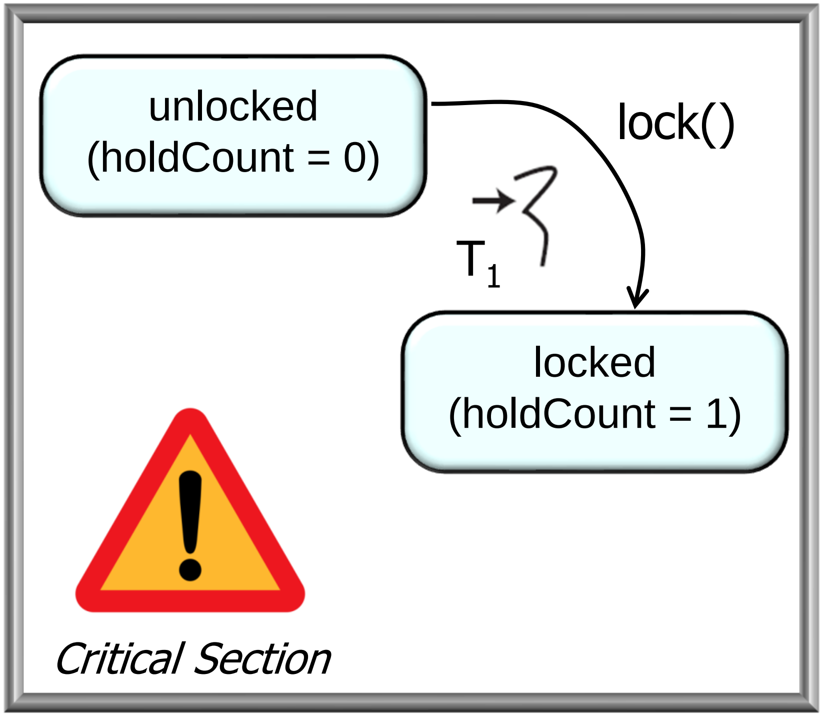
void	lock() – Acquires the lock
void	unlock() – Attempts to release this lock
void	lockInterruptibly() – Acquires the lock unless the current thread is interrupted
boolean	tryLock() – Acquires the lock only if it is not held by another thread at the time of invocation
boolean	tryLock(long timeout, Timeunit 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

Likewise, Java's synchronized methods/statements aren't non-blocking

Overview of Reentrant Mutex Semantics

Overview of Reentrant Mutex Semantics

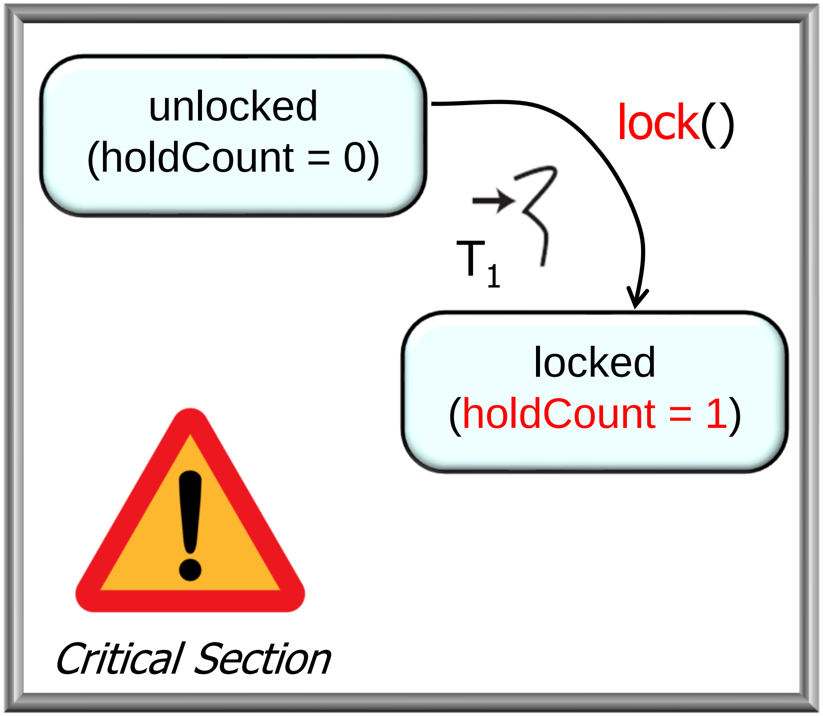
- A ReentrantLock supports "reentrant mutex" semantics



See en.wikipedia.org/wiki/Reentrant_mutex

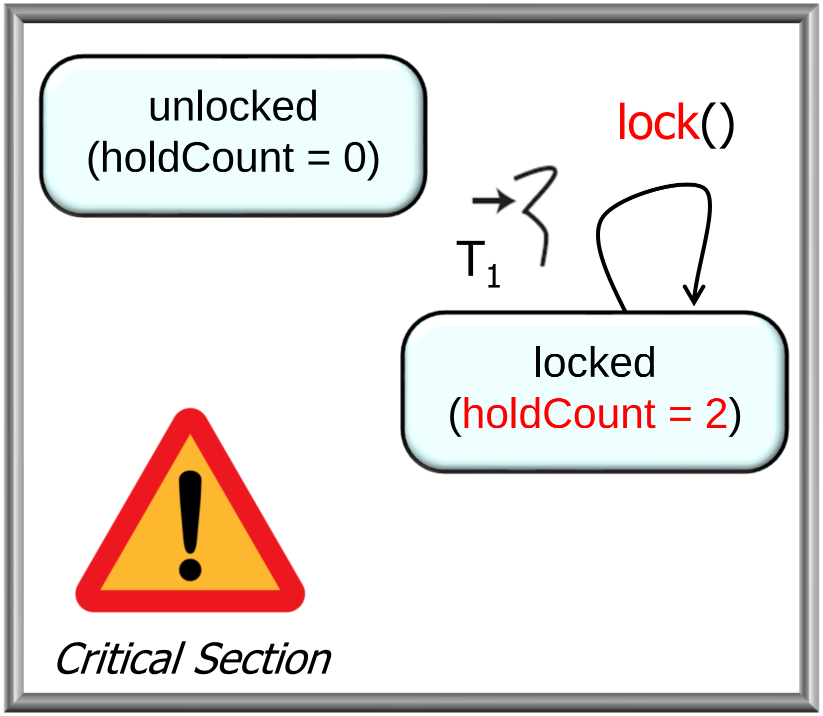
Overview of Reentrant Mutex Semantics

- A ReentrantLock supports "reentrant mutex" semantics
  - The thread that hold the mutex can reacquire it without self-deadlock



Overview of Reentrant Mutex Semantics

- A ReentrantLock supports "reentrant mutex" semantics
  - The thread that hold the mutex can reacquire it without self-deadlock



Overview of Reentrant Mutex Semantics

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

```
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

Overview of Reentrant Mutex Semantics

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

```
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 read the current hold count

Overview of Reentrant Mutex Semantics

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

```
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

Overview of Reentrant Mutex Semantics

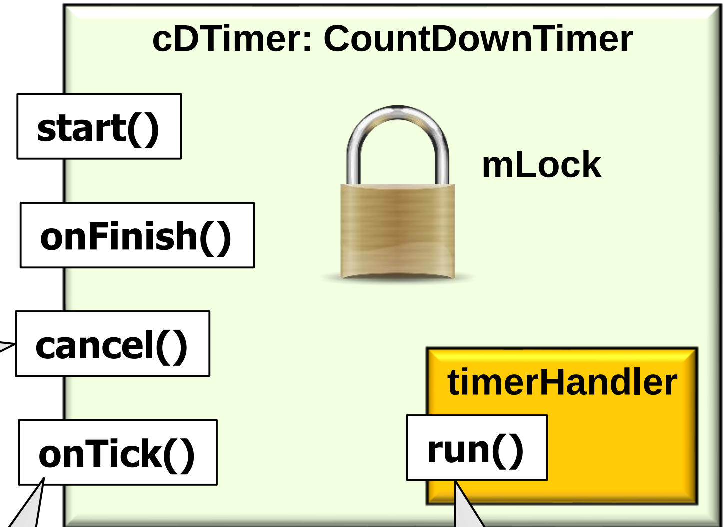
- Reentrant mutex semantics add a bit more overhead relative to non-recursive semantics due to extra software logic & synchronization

```
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;
}
```

Simply increment lock count if the current thread is lock owner

Overview of Reentrant Mutex Semantics

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




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

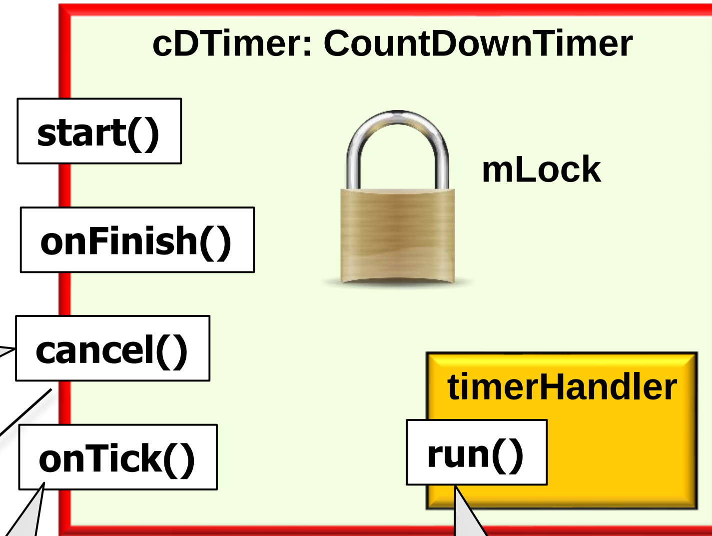
```
if (...)
  cancel();
```

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

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex24

Overview of Reentrant Mutex Semantics

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




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

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

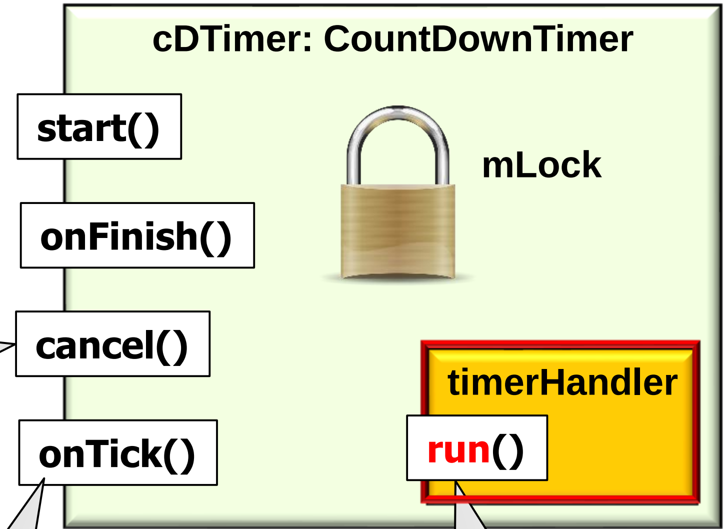
```
mLock.lock();
try {
  ...
  onTick(millisLeft);
  ...
} finally {
  mLock.unlock();
}
```

Schedule a countdown until a time in the future, with regular notifications on intervals along the way via the onTick() hook method

Overview of Reentrant Mutex Semantics

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

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




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

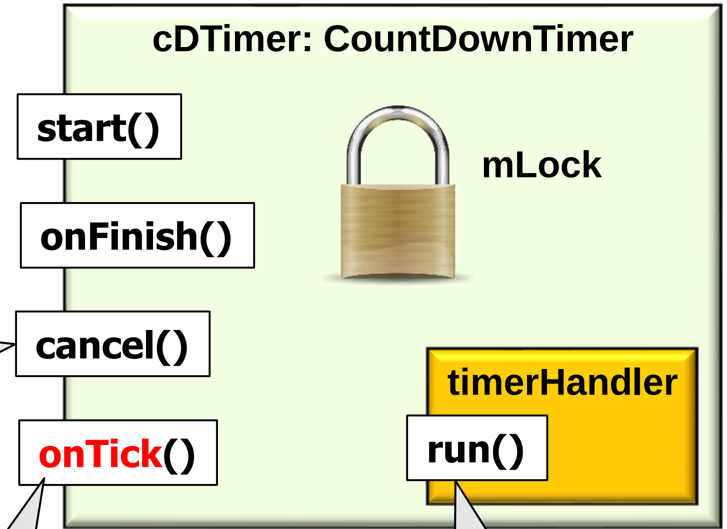
```
mLock.lock();
try {
  ...
  onTick(millisLeft);
  ...
} finally {
  mLock.unlock();
}
```

Framework calls onTick() hook method in a background thread with the mLock held

Overview of Reentrant Mutex Semantics

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

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



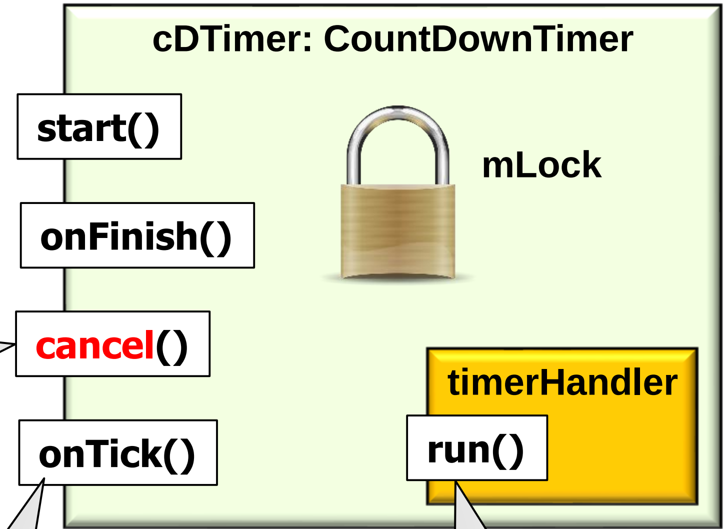

```
if (...)
   cancel();
```

The app can override the onTick() hook method to conditionally call cancel()

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

Overview of Reentrant Mutex Semantics

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




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

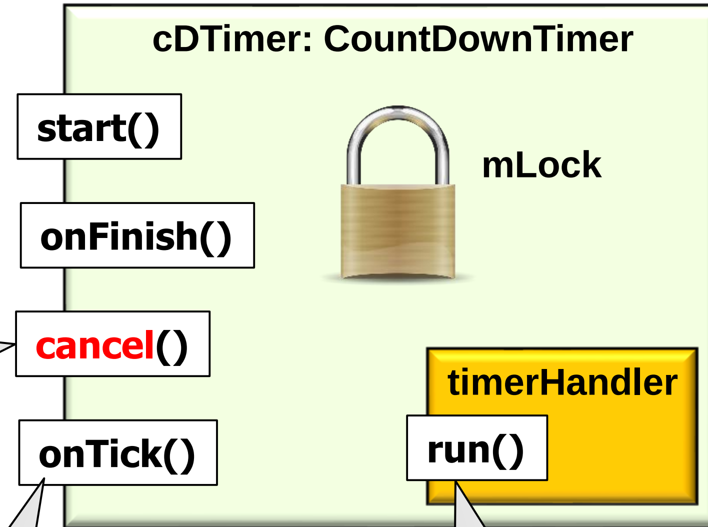
```
if (...)
  cancel();
```

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

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

Overview of Reentrant Mutex Semantics

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




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

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

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

unlock() will be called multiple times to unwind the reentrant lock

End of Java ReentrantLock: Structure & Functionality