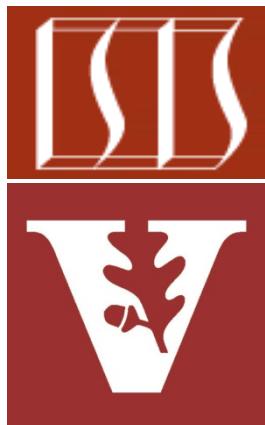


# **Java ReentrantLock**

# **Reentrant Mutex Semantics**



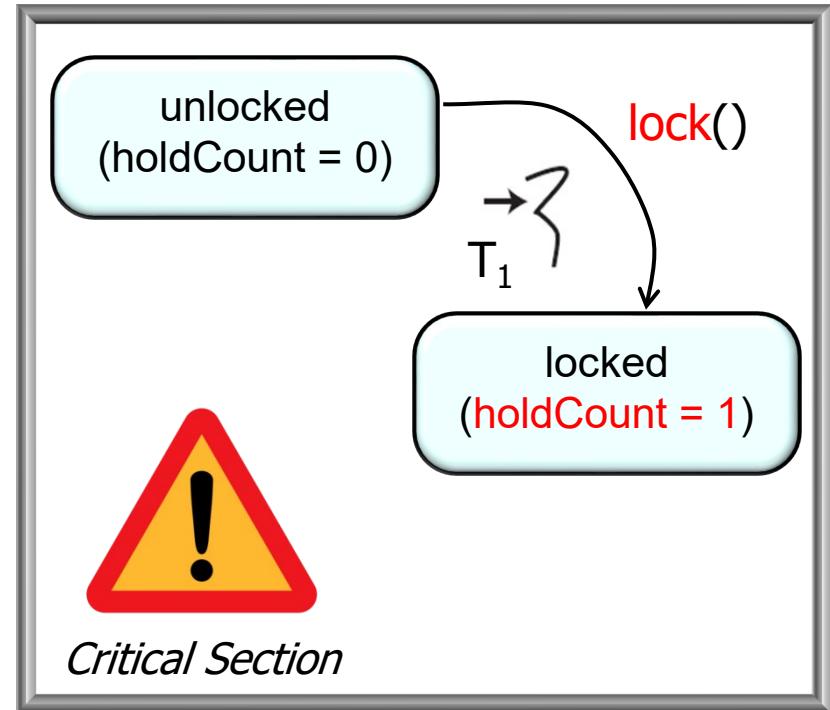
**Douglas C. Schmidt**  
[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)  
[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)

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



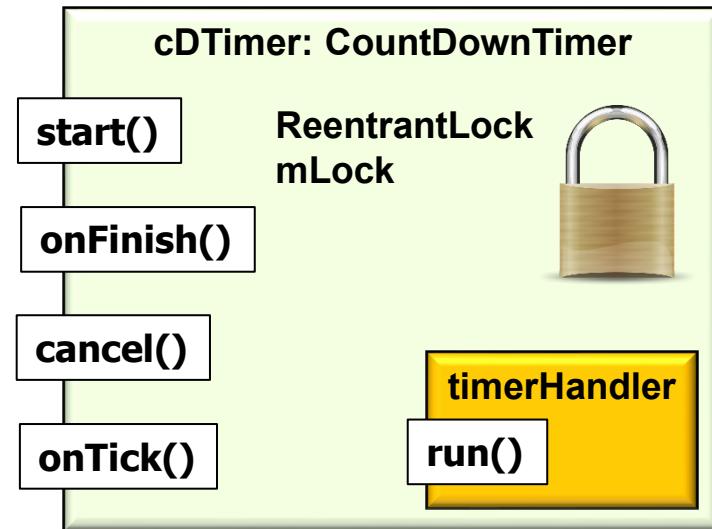
# Learning Objectives in this Part of the Lesson

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



# 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
  - In the context of Java ReentrantLock & the Android CountDownTimer framework



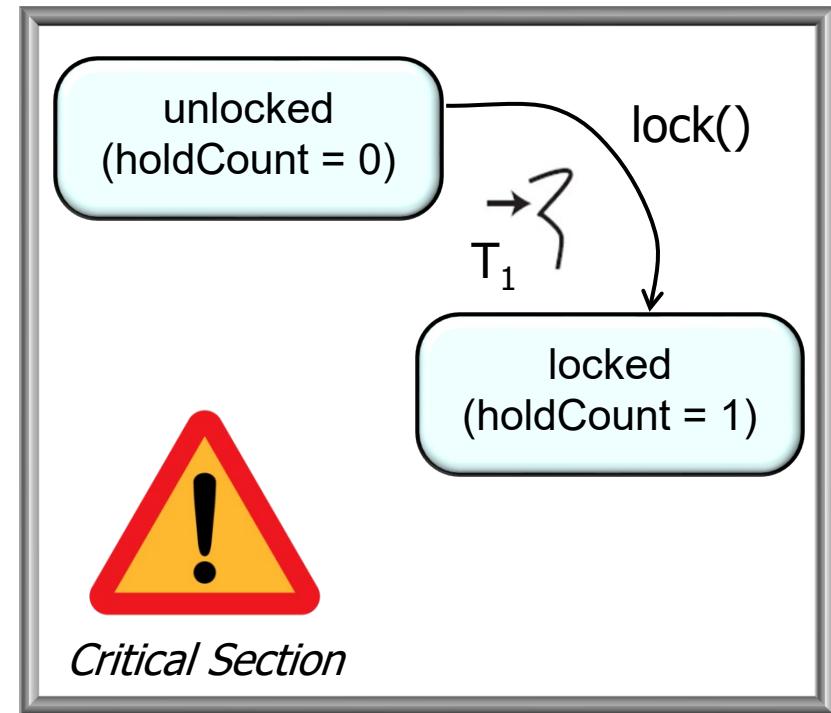
See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex24](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex24)

---

# Overview of Reentrant Mutex Semantics

# Overview of Reentrant Mutex Semantics

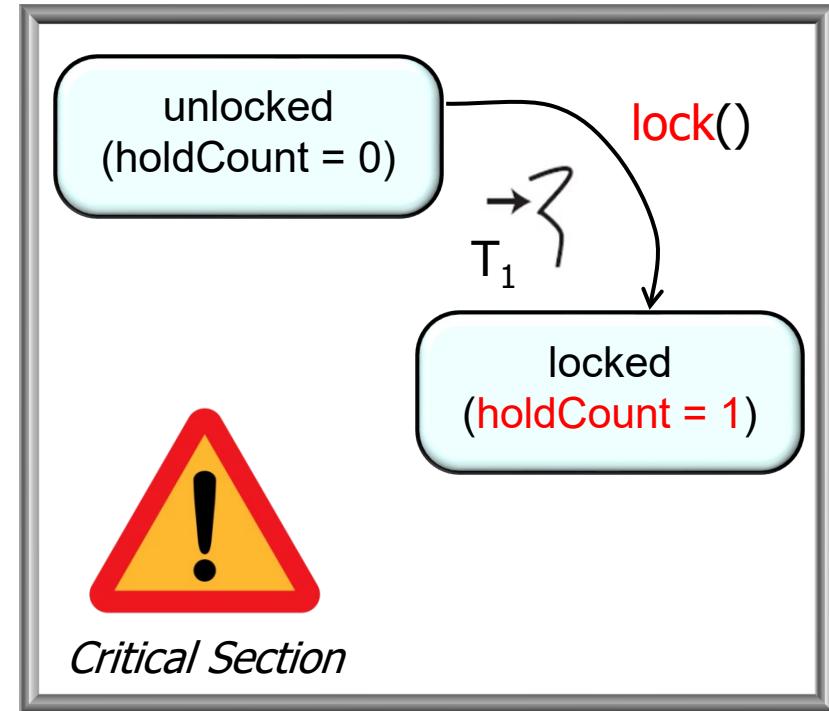
- A ReentrantLock supports “reentrant mutex” semantics



See [en.wikipedia.org/wiki/Reentrant\\_mutex](https://en.wikipedia.org/wiki/Reentrant_mutex)

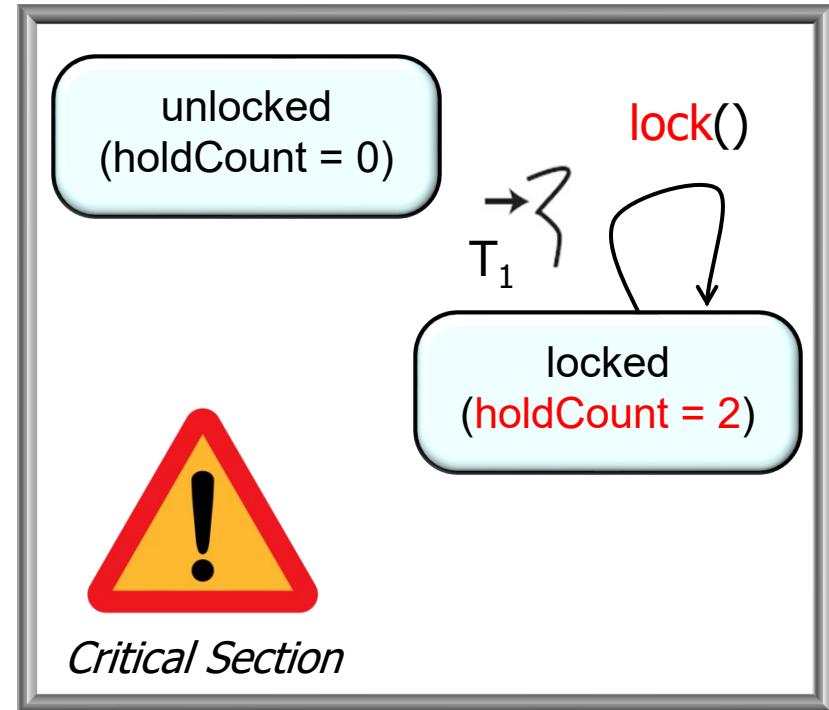
# Overview of Reentrant Mutex Semantics

- A ReentrantLock supports “reentrant mutex” semantics
  - The thread holding the mutex can reacquire it without self-deadlock



# Overview of Reentrant Mutex Semantics

- A ReentrantLock supports “reentrant mutex” semantics
  - The thread holding 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

*Record the calling thread identity*

```
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;  
}
```

# Overview of Reentrant Mutex Semantics

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

*Atomically read the current hold count*

```
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;  
}
```

# Overview of Reentrant Mutex Semantics

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

*Atomically acquire the lock if it's available*

```
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;  
}
```

# Overview of Reentrant Mutex Semantics

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

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

```
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;  
}
```

# Overview of Reentrant Mutex Semantics

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

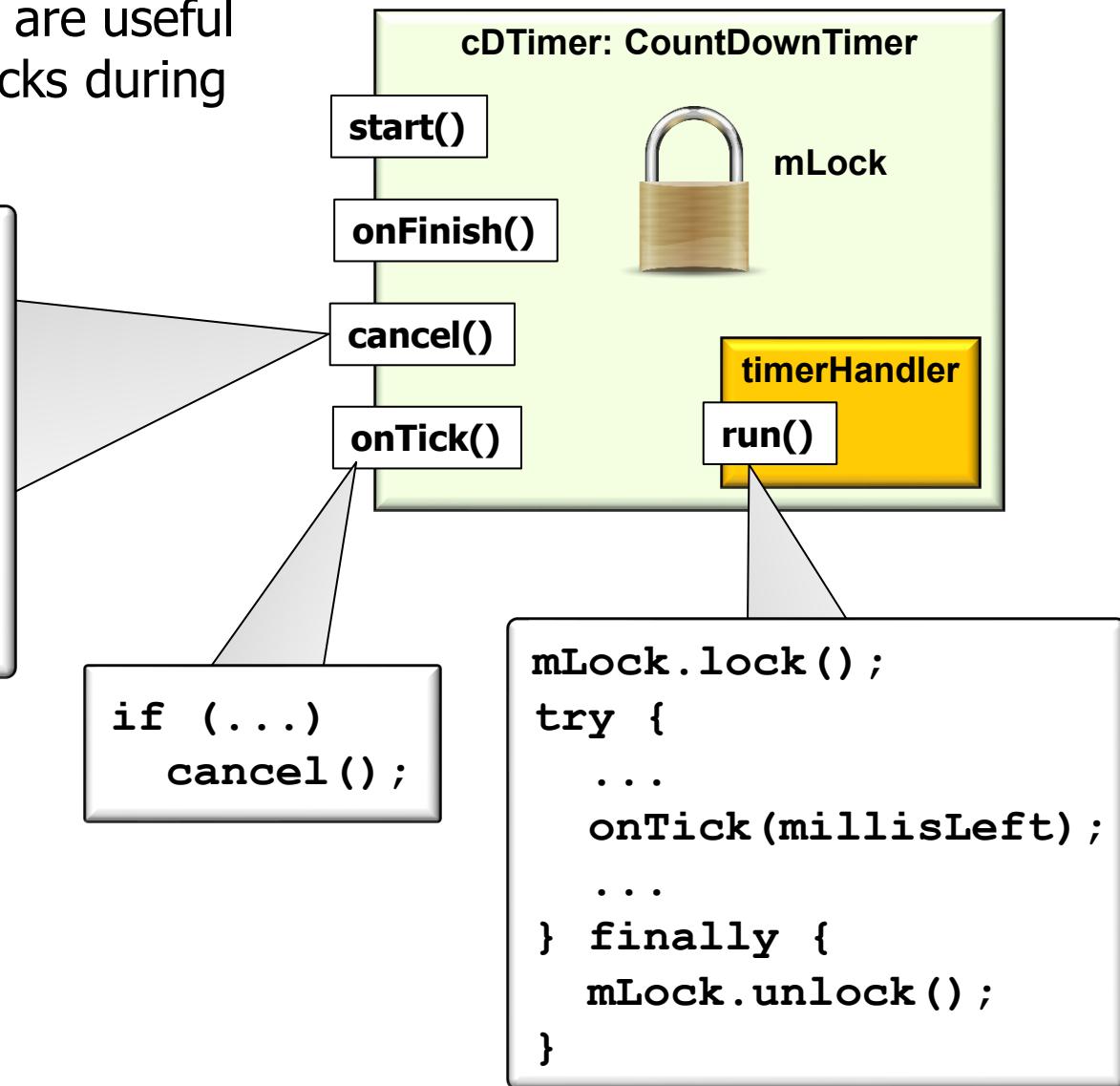
*Return false if the calling thread doesn't own the lock*

```
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;  
}
```

# Overview of Reentrant Mutex Semantics

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

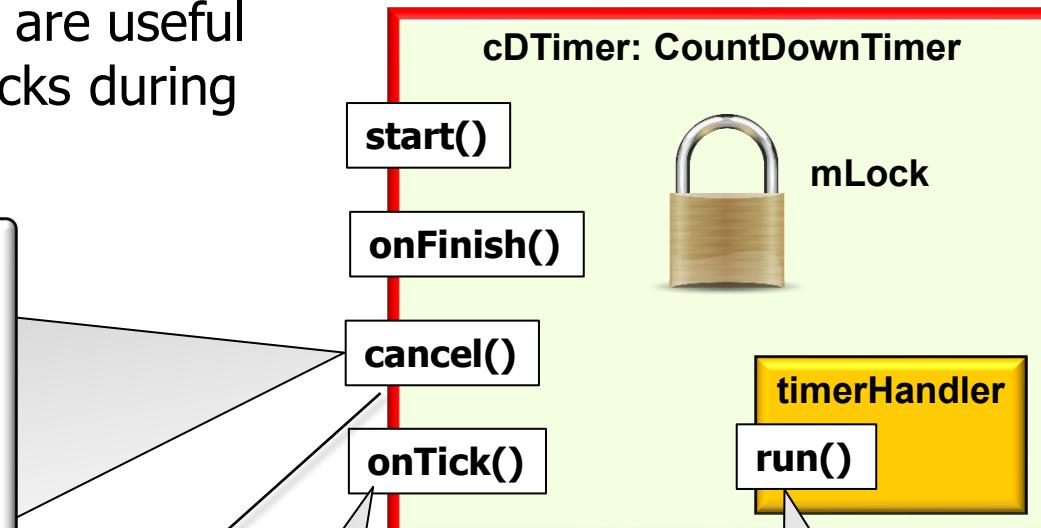
```
mLock.lock();  
try {  
    mCancelled = true;  
    mSchedExeSvc  
        .shutdownNow();  
} finally {  
    mLock.unlock();  
}
```



# 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();  
}
```



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

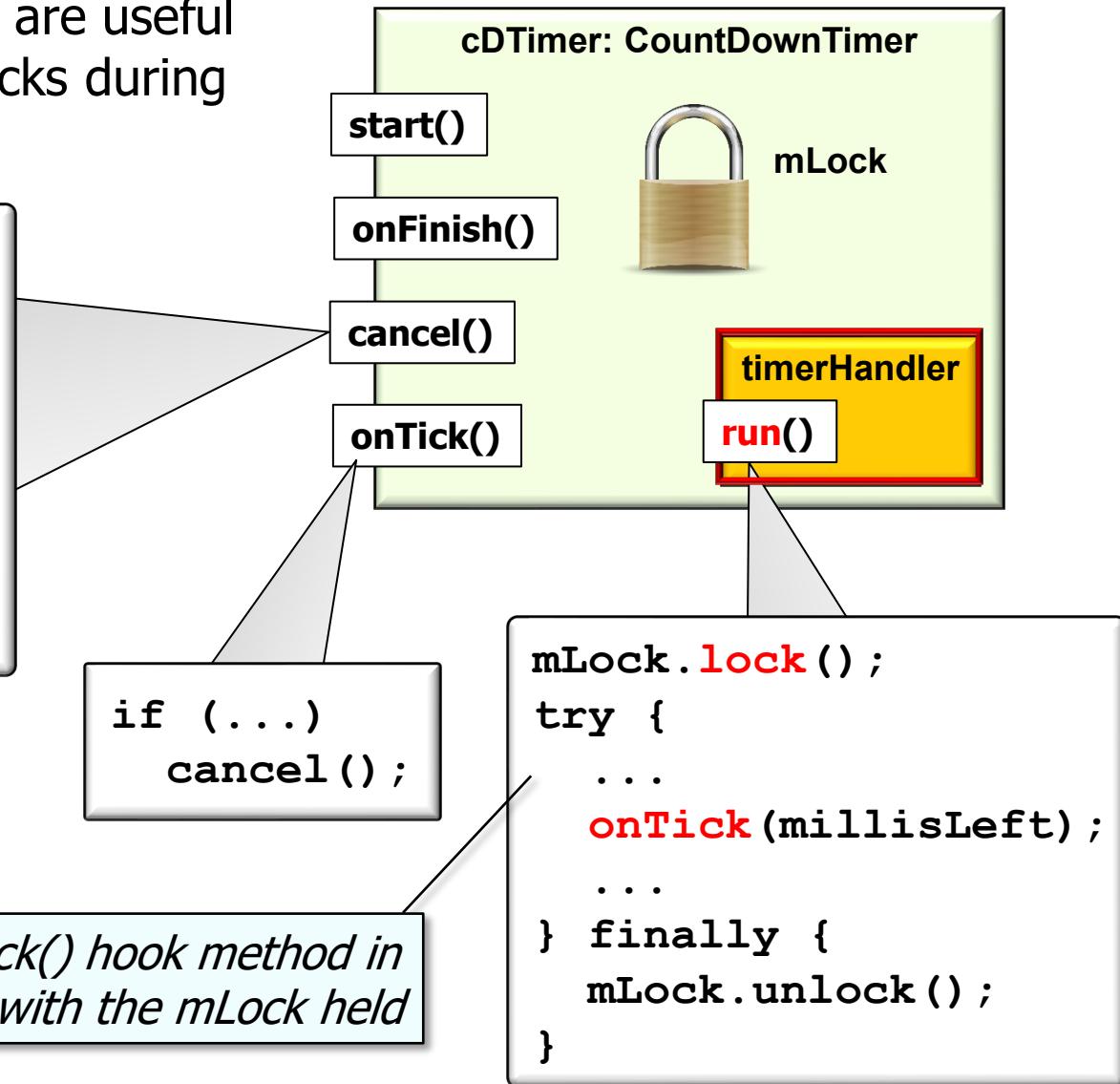
```
mLock.lock();  
try {  
    ...  
    onTick(millisLeft);  
    ...  
} finally {  
    mLock.unlock();  
}
```

See [developer.android.com/reference/android/os/CountDownTimer](http://developer.android.com/reference/android/os/CountDownTimer)

# Overview of Reentrant Mutex Semantics

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

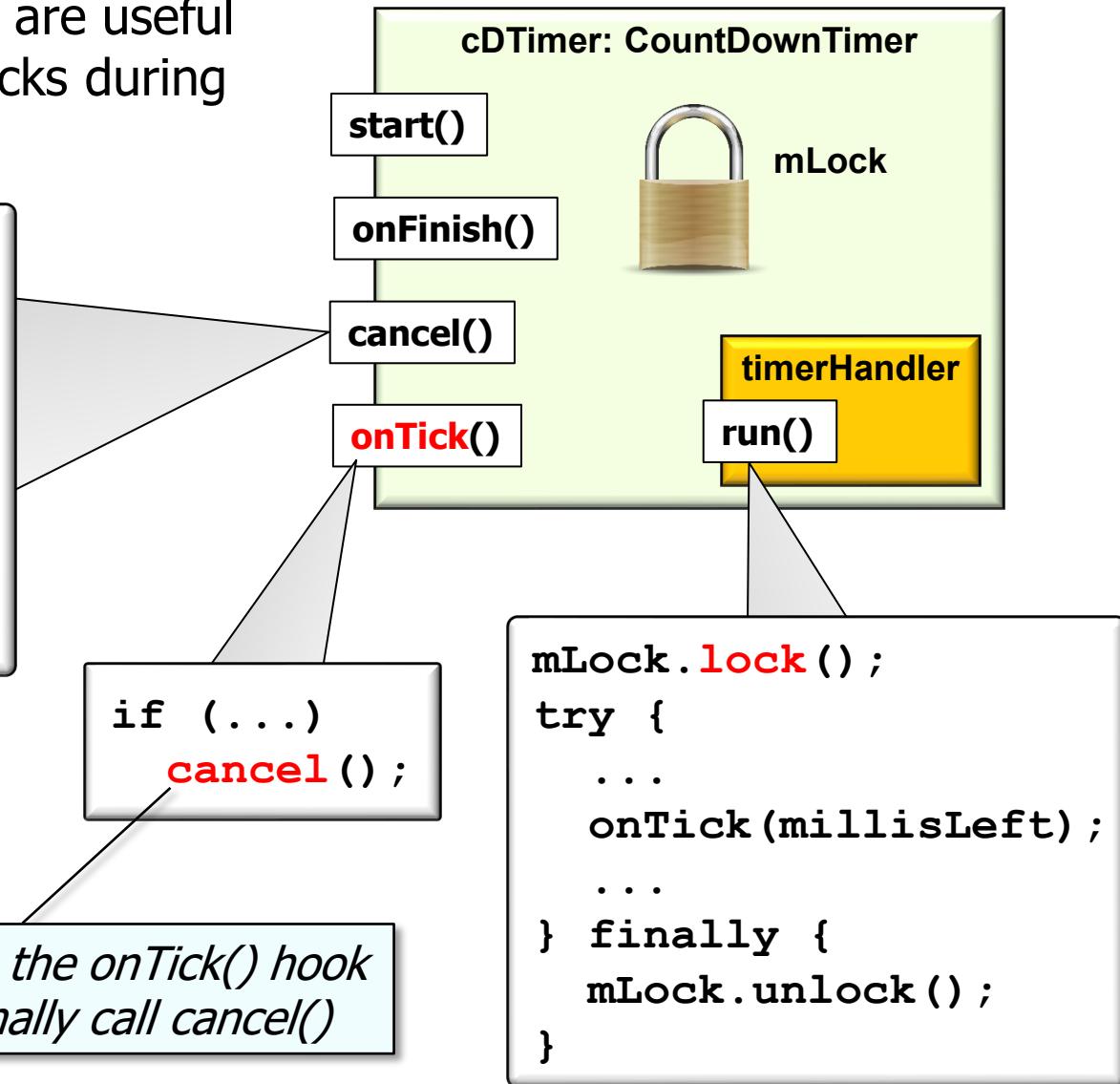
```
mLock.lock();  
try {  
    mCancelled = true;  
    mSchedExeSvc  
        .shutdownNow();  
} finally {  
    mLock.unlock();  
}
```



# Overview of Reentrant Mutex Semantics

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

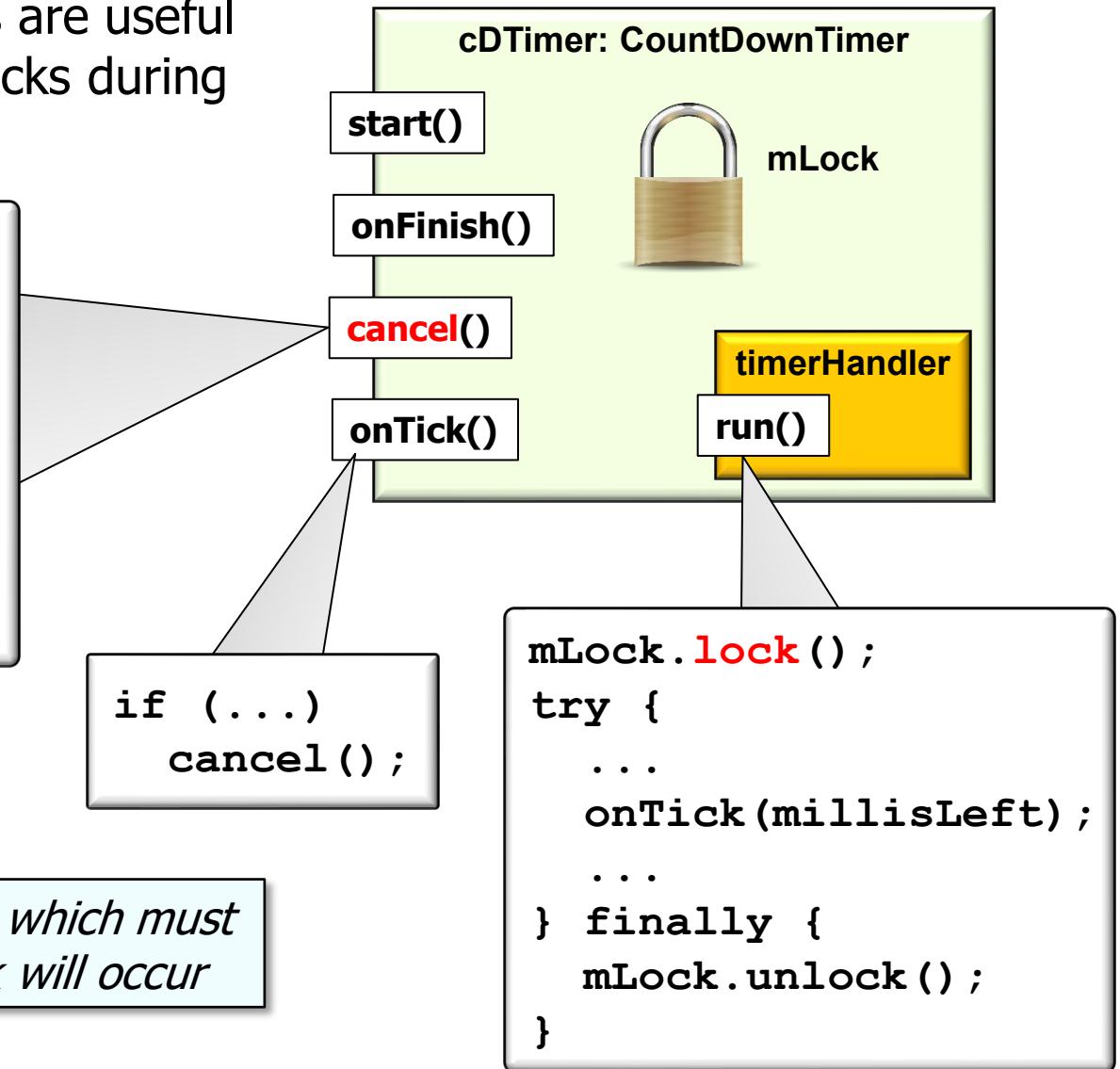
```
mLock.lock();  
try {  
    mCancelled = true;  
    mSchedExeSvc  
        .shutdownNow();  
} finally {  
    mLock.unlock();  
}
```



# 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();  
}
```

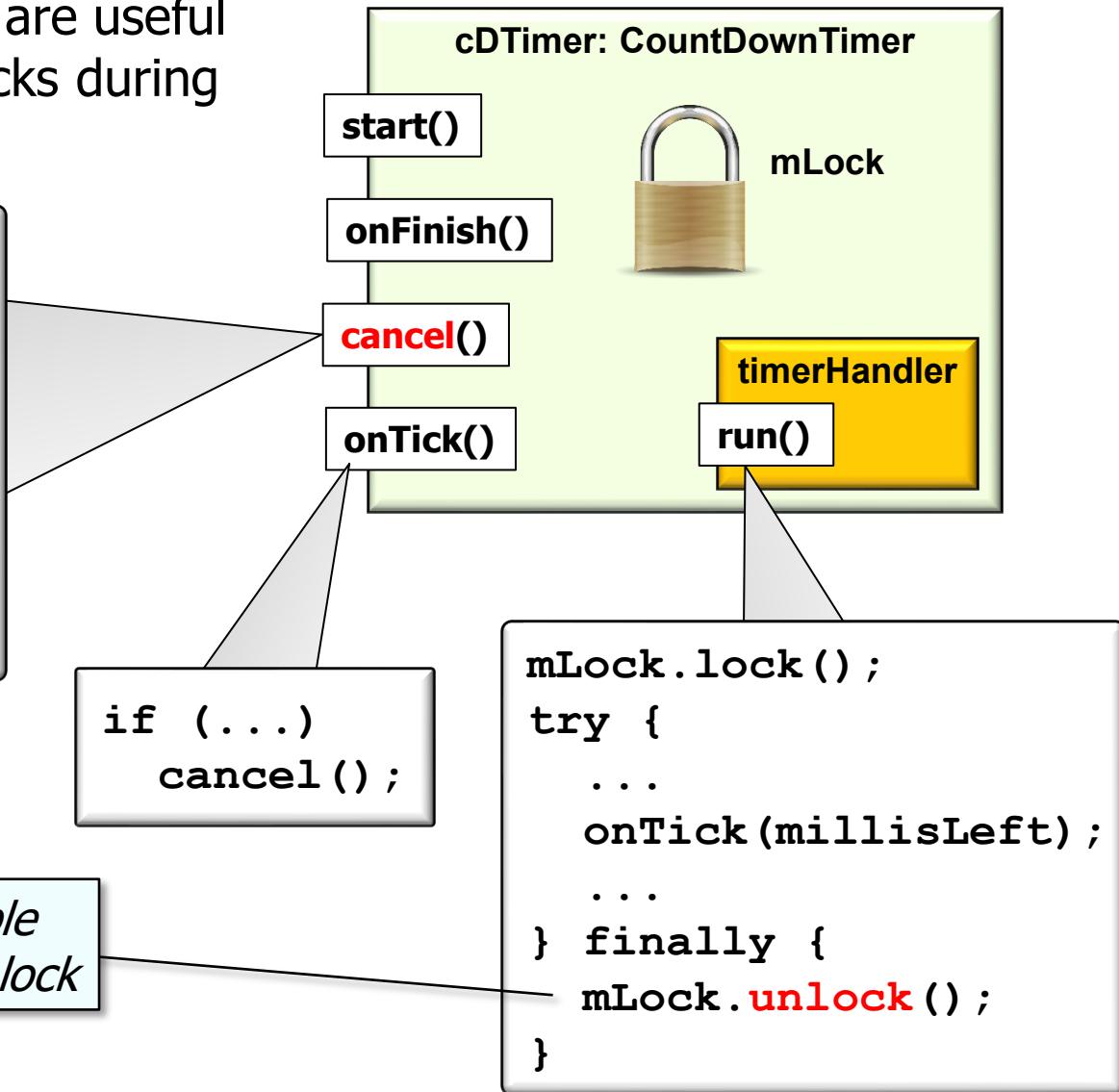


# Overview of Reentrant Mutex Semantics

- 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();  
}
```

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



# Overview of Reentrant Mutex Semantics

The screenshot shows a Java code editor in an IDE. The project structure on the left shows a directory named 'ex24' containing '.idea', '.settings', 'out', 'src', 'utils', 'CountDownTimer.java', 'NonReentrantLock.java', 'ex24.java', '.classpath', '.project', and 'ex24.iml'. The 'ex24.java' file is open in the editor.

```
12  /**
13   * This example shows the difference between a reentrant lock (e.g.,
14   * Java ReentrantLock) and a non-reentrant lock (e.g., Java
15   * StampedLock) when applied in a framework that allows callbacks
16   * where the framework holds a lock protecting internal framework state.
17   * As you'll see when you run this program, the reentrant lock supports
18   * this use-case nicely, whereas the non-reentrant lock incurs
19   * "self-deadlock."
20  */
21 > public class ex24 {
22  /**
23   * Used to wait for the test to finish running.
24  */
25  private static CountDownLatch sCdl;
26
27  /**
28   * Main entry point into the test program.
29  */
30 > public static void main (String[] argv)
31  throws IOException, InterruptedException {
```

The code is annotated with comments explaining the purpose of the reentrant lock and the non-reentrant lock. It includes a main method that uses a CountDownLatch.

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex24](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex24)

---

# End of Java ReentrantLock Reentrant Mutex Semantics