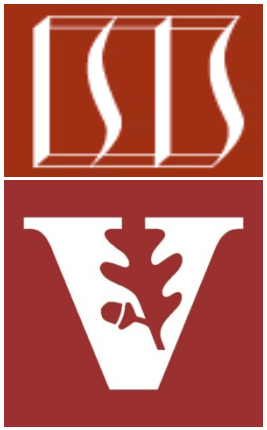
# Android Concurrency: Java ReentrantLock



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 Module

- Understand how ReentrantLocks provide mutual exclusion to concurrent Java programs

## ReentrantLock

Added in API level 1

extends Object
implements Serializable Lock

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

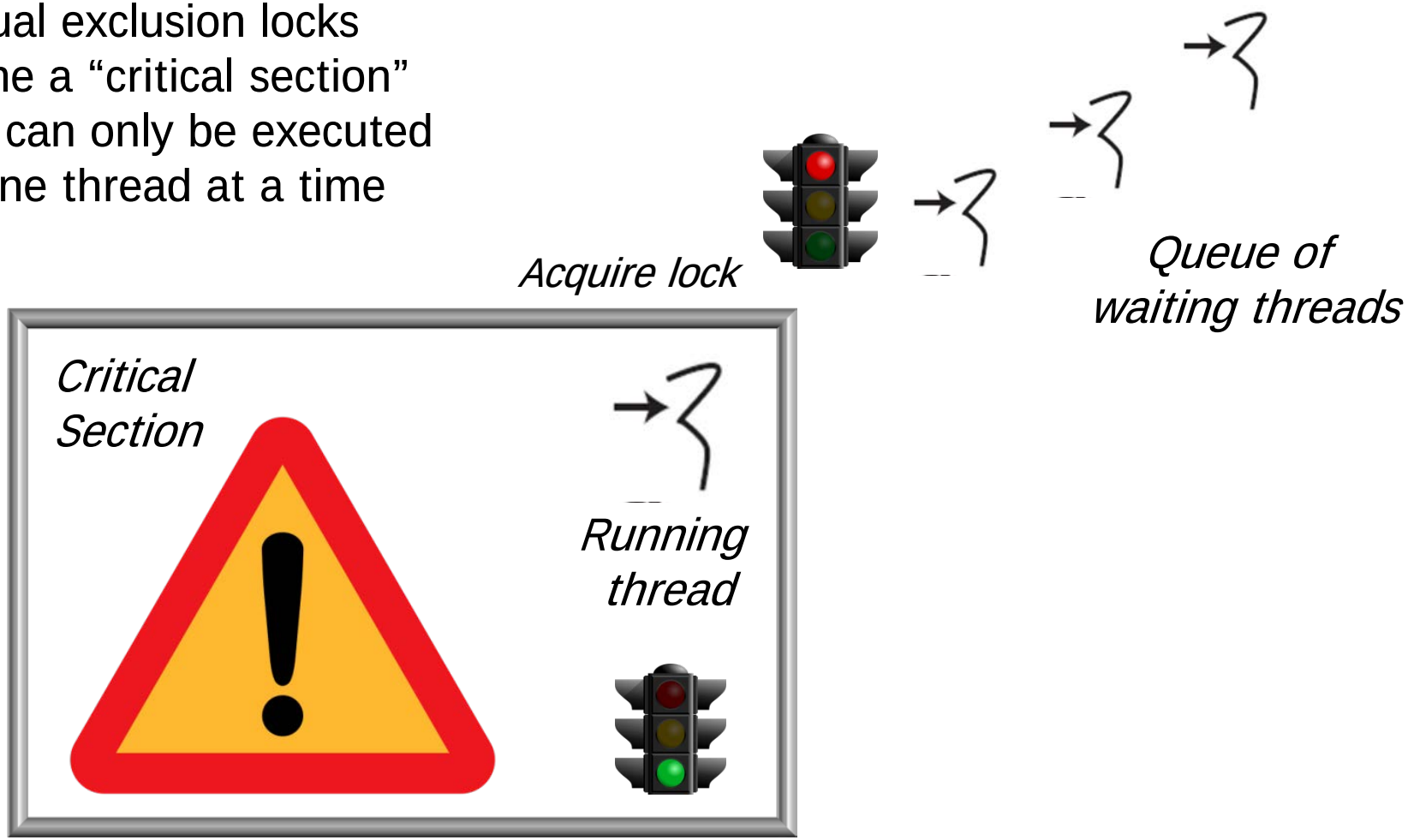
### Class Overview

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()`.

# Overview of Mutual Exclusion Locks

- Mutual exclusion locks define a “critical section” that can only be executed by one thread at a time



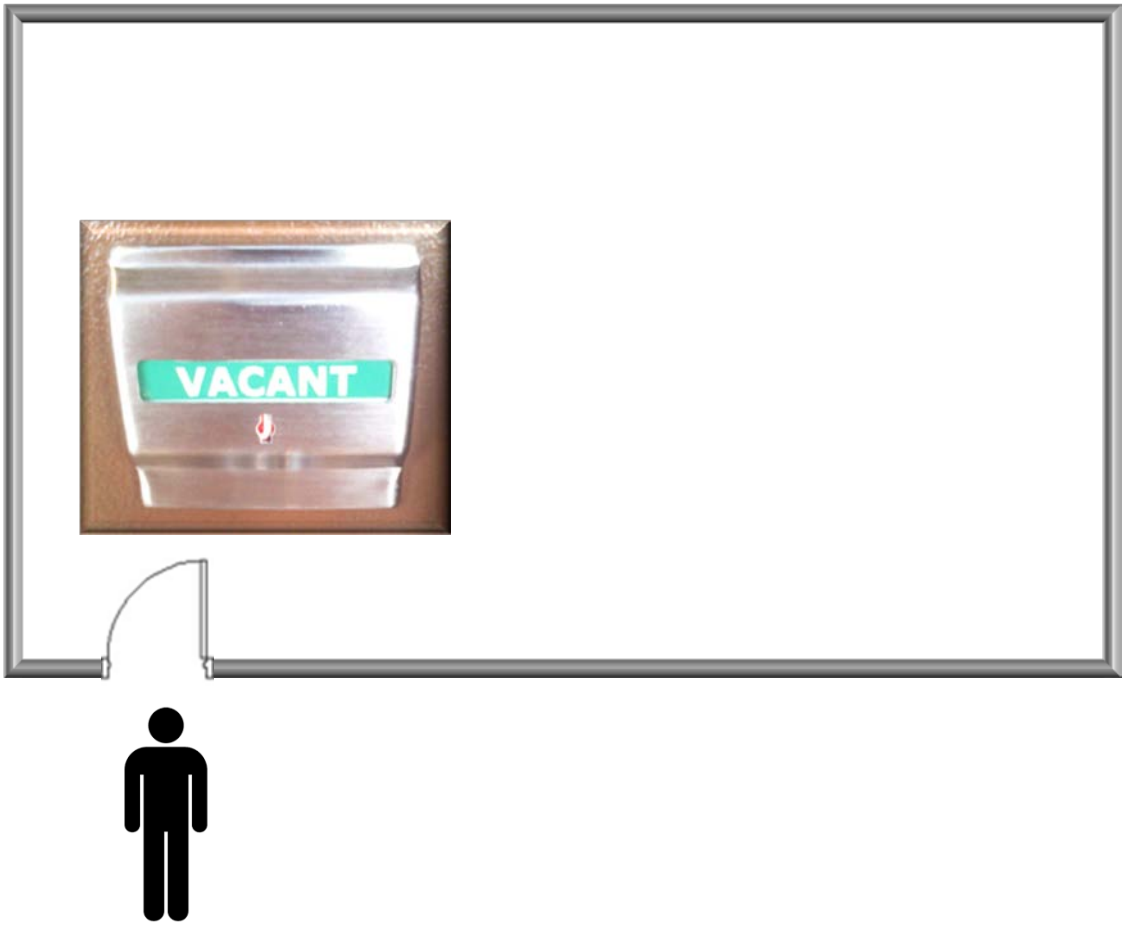

en.wikipedia.org/wiki/Mutual_exclusion has more on mutual exclusion

# Overview of Mutual Exclusion Locks

- Mutual exclusion locks define a “critical section” that can only be executed by one thread at a time
- A human known use of mutual exclusion is an airplane restroom protocol

# Overview of Mutual Exclusion Locks

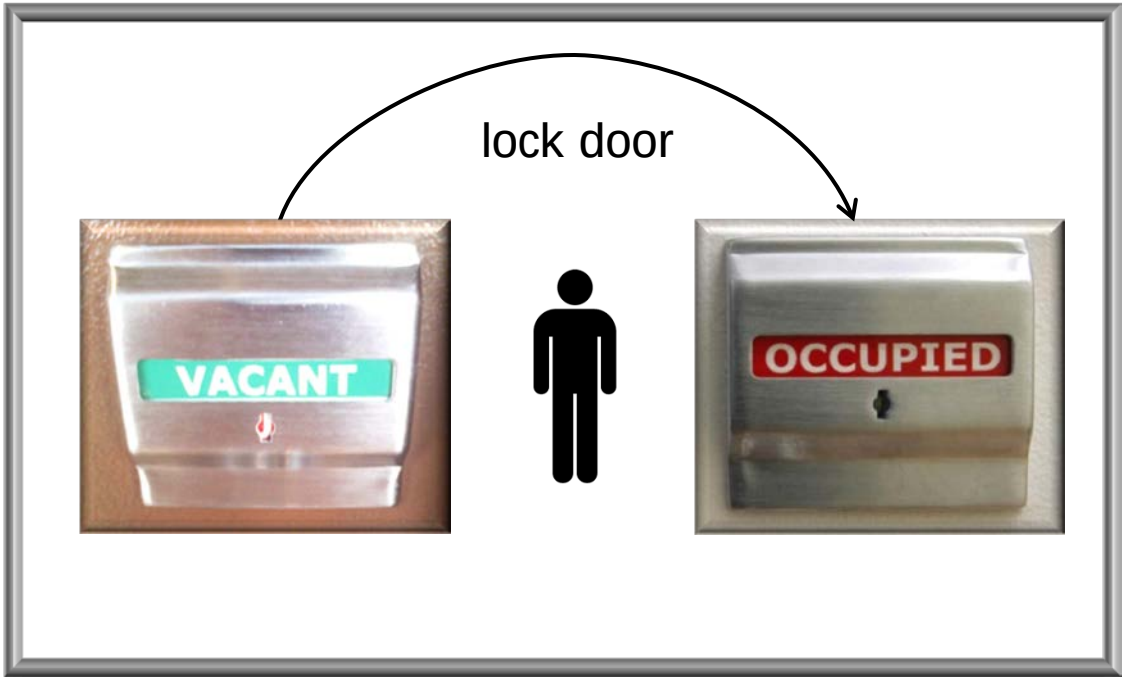
- Mutual exclusion locks define a “critical section” that can only be executed by one thread at a time
- A human known use of mutual exclusion is an airplane restroom protocol




See libcore/luni/src/main/java/java/util/concurrent/locks/ReentrantLock.java

# Overview of Mutual Exclusion Locks

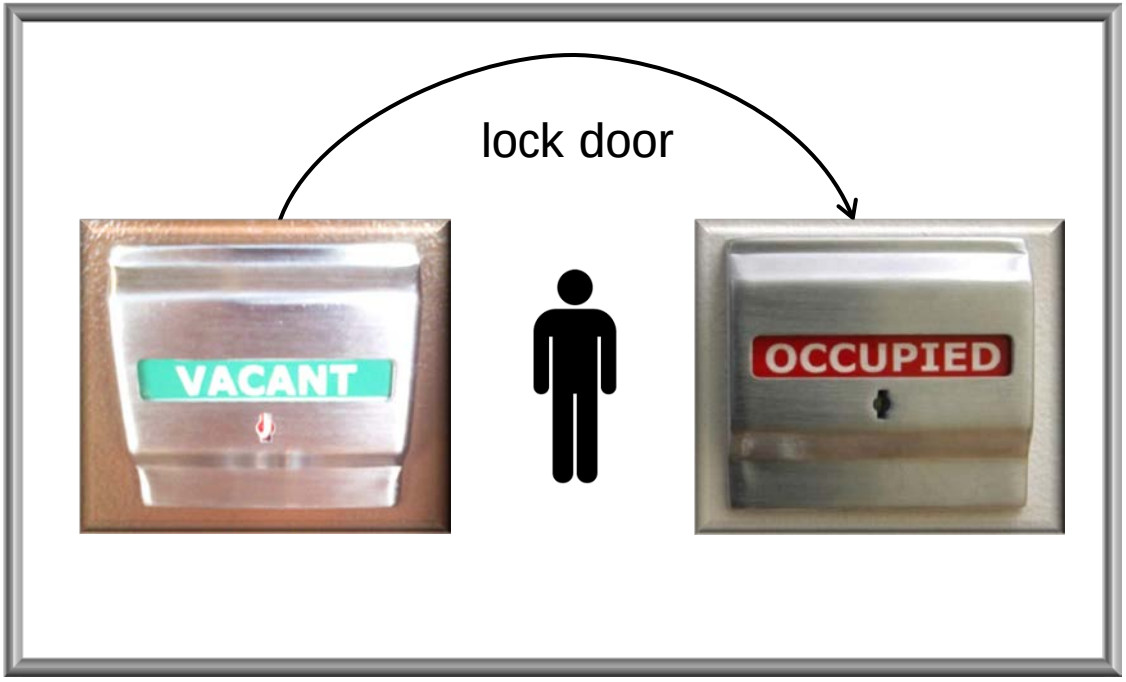
- Mutual exclusion locks define a “critical section” that can only be executed by one thread at a time
- A human known use of mutual exclusion is an airplane restroom protocol

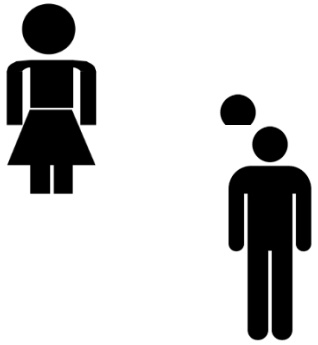



See libcore/luni/src/main/java/java/util/concurrent/locks/ReentrantLock.java

# Overview of Mutual Exclusion Locks

- Mutual exclusion locks define a “critical section” that can only be executed by one thread at a time
- A human known use of mutual exclusion is an airplane restroom protocol

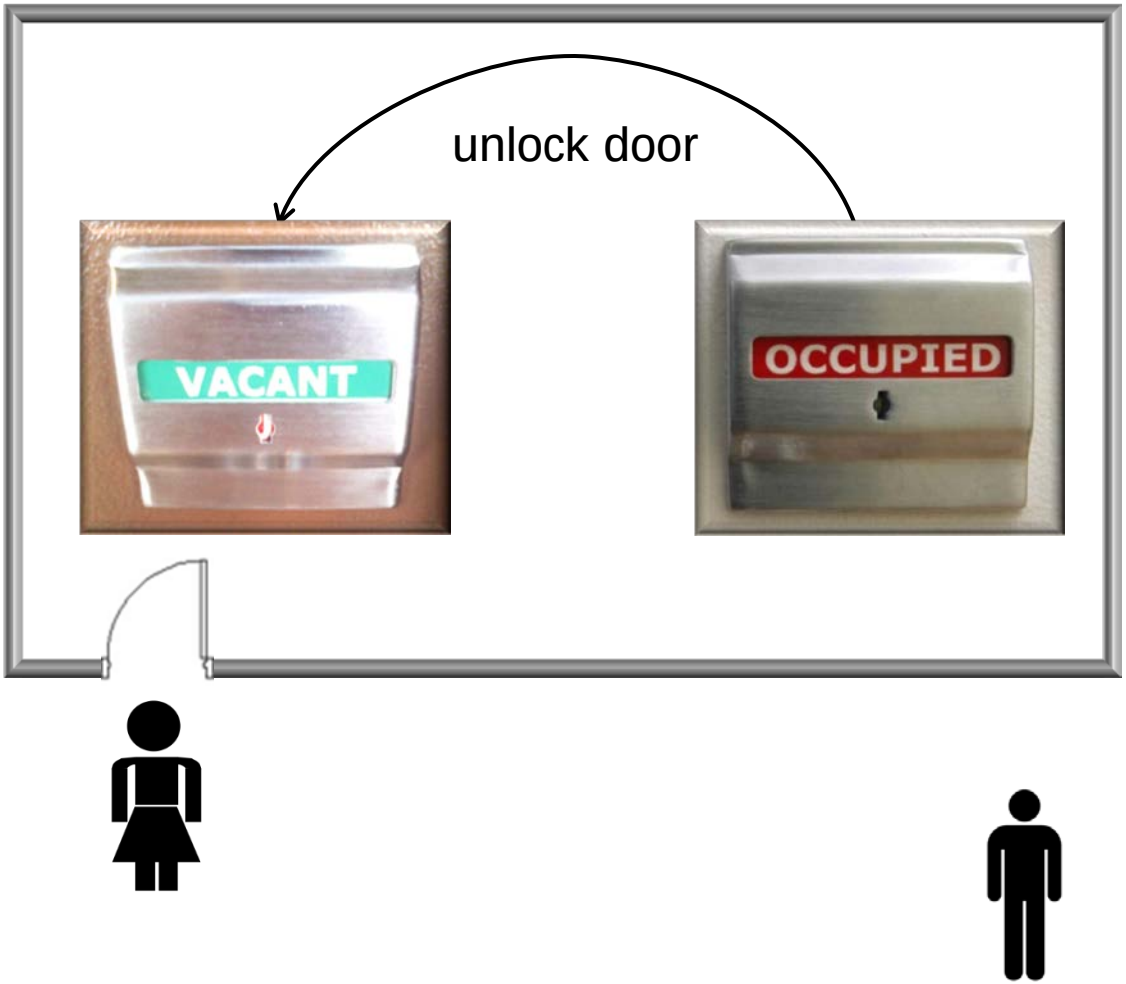


See libcore/luni/src/main/java/java/util/concurrent/locks/ReentrantLock.java

# Overview of Mutual Exclusion Locks

- Mutual exclusion locks define a “critical section” that can only be executed by one thread at a time
- A human known use of mutual exclusion is an airplane restroom protocol



# Overview of ReentrantLock

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion

## ReentrantLock

Added in API level 1

extends Object
implements Serializable Lock

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

### Class Overview

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()`.

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
  - Most of it is written in Java

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

  ...
```

See libcore/luni/src/main/java/java/util/concurrent/locks/ReentrantLock.java

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
- ReentrantLock uses the *Bridge* pattern

```
public class ReentrantLock
             implements Lock,
             java.io.Serializable {
  ...
  /** Synchronizer providing all
      implementation mechanics */
  private final Sync sync;

  /**
    * Synchronization imp
    * for semaphore...
    */
  abstract static class Sync
    extends
    AbstractQueuedSynchronizer
    {
      ...
    }
```

en.wikipedia.org/wiki/Bridge_pattern contains more on the *Bridge* pattern

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
- ReentrantLock uses the *Bridge* pattern
  - Inherits functionality from the AbstractQueuedSynchronizer class

```
public class ReentrantLock
             implements Lock,
             java.io.Serializable {
  ...
  /** Synchronizer providing all
      implementation mechanics */
  private final Sync sync;

  /**
    * Synchronization imp
    * for semaphore...
    */
  abstract static class Sync
    extends
    AbstractQueuedSynchronizer
    {
      ...
    }
```

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
- ReentrantLock uses the *Bridge* pattern
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model

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

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

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
- ReentrantLock uses the *Bridge* pattern
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model

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

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

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
- ReentrantLock uses the *Bridge* pattern
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model

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

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

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
- ReentrantLock uses the *Bridge* pattern
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model

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

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

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
- ReentrantLock uses the *Bridge* pattern
- It key methods are variants of lock() & unlock()

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

  /** Acquires lock unless current
      thread is interrupted */
  public void lockInterruptibly()
      throws InterruptedException {
     sync.acquireInterruptibly(1);
  }

  /** Release the lock. */
  public void unlock() {
    sync.release(1);
  }
  ...
```

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
- ReentrantLock uses the *Bridge* pattern
- It key methods are variants of lock() & unlock()

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

  /** Acquires lock unless current
      thread is interrupted */
  public void lockInterruptibly()
      throws InterruptedException {
     sync.acquireInterruptibly(1);
  }

  /** Release the lock. */
  public void unlock() {
    sync.release(1);
  }
  ...
```

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
- ReentrantLock uses the *Bridge* pattern
- It key methods are variants of lock() & unlock()
  - lock() acquires the lock if it's available

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

  /** Acquires lock unless current
      thread is interrupted */
  public void lockInterruptibly()
      throws InterruptedException {
     sync.acquireInterruptibly(1);
  }

  /** Release the lock. */
  public void unlock() {
    sync.release(1);
  }
  ...
```

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
- ReentrantLock uses the *Bridge* pattern
- It key methods are variants of lock() & unlock()
  - lock() acquires the lock if it's available
  - lockInterruptibly() acquires the lock unless thread is interrupted

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

  /** Acquires lock unless current
      thread is interrupted */
  public void lockInterruptibly()
      throws InterruptedException {
     sync.acquireInterruptibly(1);
  }

  /** Release the lock. */
  public void unlock() {
    sync.release(1);
  }
  ...
```

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
- ReentrantLock uses the *Bridge* pattern
- It key methods are variants of lock() & unlock()
  - lock() acquires the lock if it's available
  - lockInterruptibly() acquires the lock unless thread is interrupted
  - unlock() attempts to release the lock

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

  /** Acquires lock unless current
      thread is interrupted */
  public void lockInterruptibly()
      throws InterruptedException {
     sync.acquireInterruptibly(1);
  }

  /** Release the lock. */
  public void unlock() {
    sync.release(1);
  }
  ...
```

# Overview of ReentrantLock

- ReentrantLock implements the Lock interface to provide mutual exclusion
- ReentrantLock uses the *Bridge* pattern
- It key methods are variants of lock() & unlock()
- These methods simply forward to their implementations
  - Non-fair implementations are optimized

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

  /** Acquires lock unless current
      thread is interrupted */
  public void lockInterruptibly()
      throws InterruptedException {
    sync.acquireInterruptibly(1);
  }

  /** Release the lock. */
  public void unlock() {
    sync.release(1);
  }
  ...
```

en.wikipedia.org/wiki/Spinlock has more on spin locks

# Using ReentrantLock in Android

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

## ArrayBlockingQueue

Added in API level 1

extends AbstractQueue<E>
implements Serializable BlockingQueue<E>

java.lang.Object
  ↳java.util.AbstractCollection<E>
    ↳java.util.AbstractQueue<E>
      ↳java.util.concurrent.ArrayBlockingQueue<E>

### Class Overview

A bounded `blocking queue` backed by an array. This queue orders elements FIFO (first-in-first-out). The *head* of the queue is that element that has been on the queue the longest time. The *tail* of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

This is a classic "bounded buffer", in which a fixed-sized array holds elements inserted by producers and extracted by consumers. Once created, the capacity cannot be changed. Attempts to `put` an element into a full queue will result in the operation blocking; attempts to `take` an element from an empty queue will similarly block.

This class supports an optional fairness policy for ordering waiting producer and consumer threads. By default, this ordering is not guaranteed. However, a queue constructed with fairness set to `true` grants threads access in FIFO order. Fairness generally decreases throughput but reduces variability and avoids starvation.

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
    implements BlockingQueue<E>,
        java.io.Serializable {
...
```

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
    implements BlockingQueue<E>,
        java.io.Serializable {
...
```

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
    implements BlockingQueue<E>,
        java.io.Serializable {
...
```

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
    implements BlockingQueue<E>,
        java.io.Serializable {
...
```

See libcore/luni/src/main/java/java/util/concurrent/ArrayBlockingQueue.java

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
    implements BlockingQueue<E>,
        java.io.Serializable {
  ...
  // Main lock guarding all access
  final ReentrantLock lock;
  ...
  // The queued items
  final Object[] items;

  // items indices for next take
  // or put calls
  int takeIndex;
  int putIndex;

  // Number of elements in the queue
  int count;
```

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
       implements BlockingQueue<E>,
              java.io.Serializable {
    ...
    // Main lock guarding all access
    final ReentrantLock lock;
    ...
    // The queued items
    final Object[] items;

    // items indices for next take
    // or put calls
    int takeIndex;
    int putIndex;

    // Number of elements in the queue
    int count;
```

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
    implements BlockingQueue<E>,
            java.io.Serializable {
  ...
  // Main lock guarding all access
  final ReentrantLock lock;
  ...
  // The queued items
  final Object[] items;

  // items indices for next take
  // or put calls
  int takeIndex;
  int putIndex;

  // Number of elements in the queue
  int count;
```

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
    implements BlockingQueue<E>,
        java.io.Serializable {
  ...
  // Main lock guarding all access
  final ReentrantLock lock;
  ...
  // The queued items
  final Object[] items;

  // items indices for next take
  // or put calls
  int takeIndex;
  int putIndex;

  // Number of elements in the queue
  int count;
```

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
         implements BlockingQueue<E>,
                java.io.Serializable {
    ...
    // Main lock guarding all access
    final ReentrantLock lock;
    ...
    // The queued items
    final Object[] items;

    // items indices for next take
    // or put calls
    int takeIndex;
    int putIndex;

    // Number of elements in the queue
    int count;
```

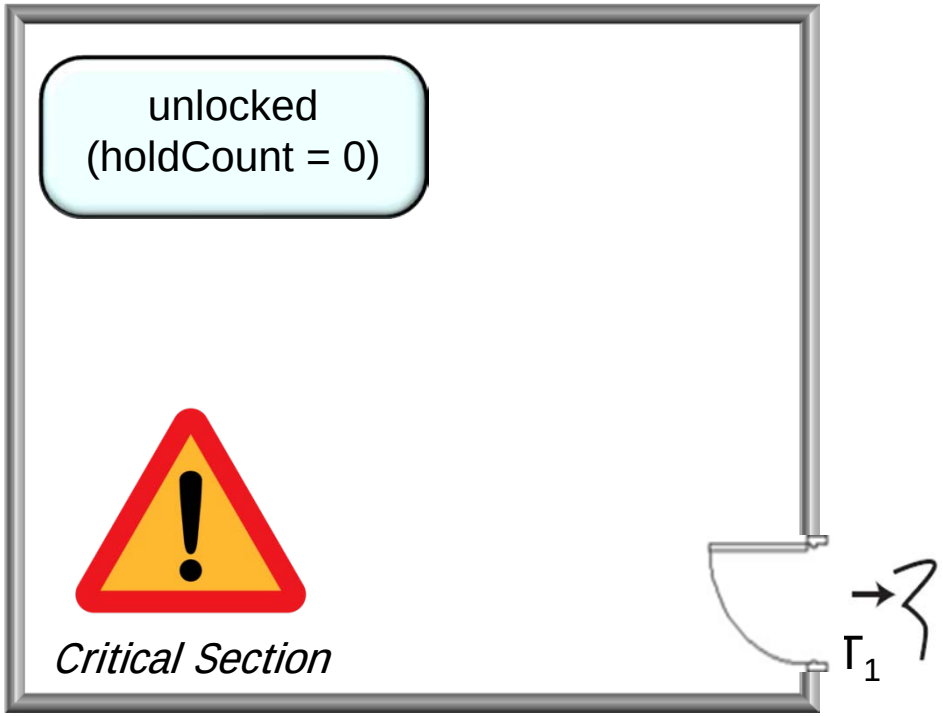
See docs.oracle.com/javase/6/docs/api/java/util/concurrent/locks/Lock.html

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile

```
ArrayBlockingQueue q = new
  ArrayBlockingQueue<String>(10);
...
// Called by thread T1
String s = q.take();
...
```

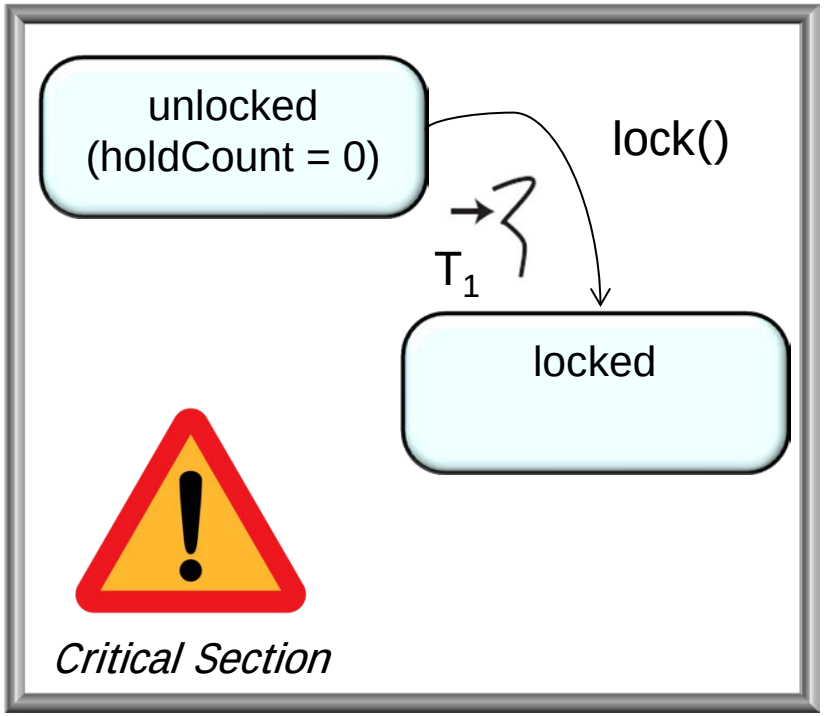
**ArrayBlockingQueue**



# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile

**ArrayBlockingQueue**

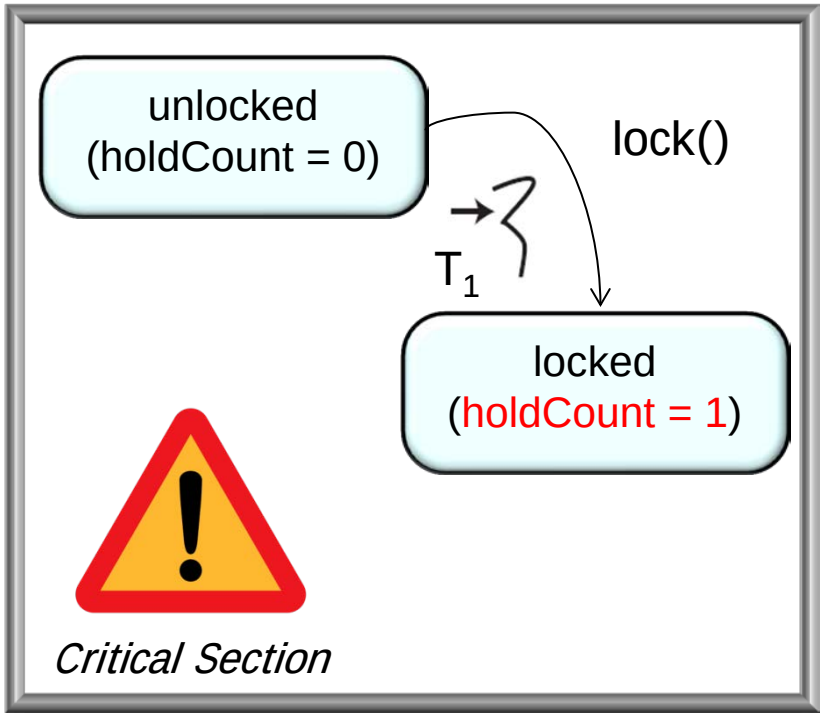



```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
         implements BlockingQueue<E>,
                java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock
      = this.lock;
    lock.lockInterruptibly();
    ...
```

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile

**ArrayBlockingQueue**

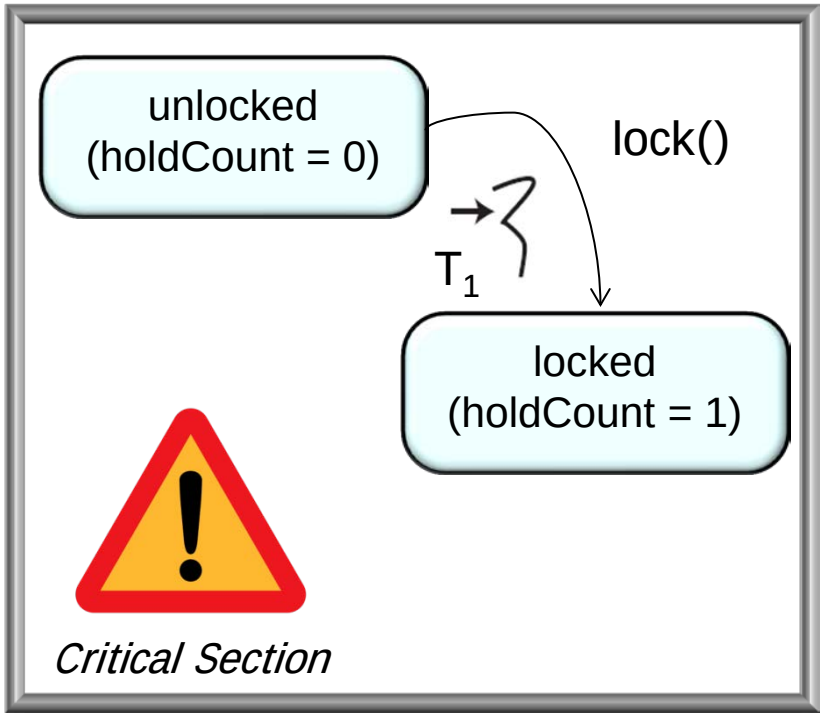



```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
         implements BlockingQueue<E>,
                java.io.Serializable {
...
public E take() ... {
  final ReentrantLock lock
    = this.lock;
  lock.lockInterruptibly();
  ...
```

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile

**ArrayBlockingQueue**




```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
         implements BlockingQueue<E>,
                java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock
      = this.lock;
    lock.lockInterruptibly();
    ...
```

*This optimization idiom produces the smallest bytecodes*

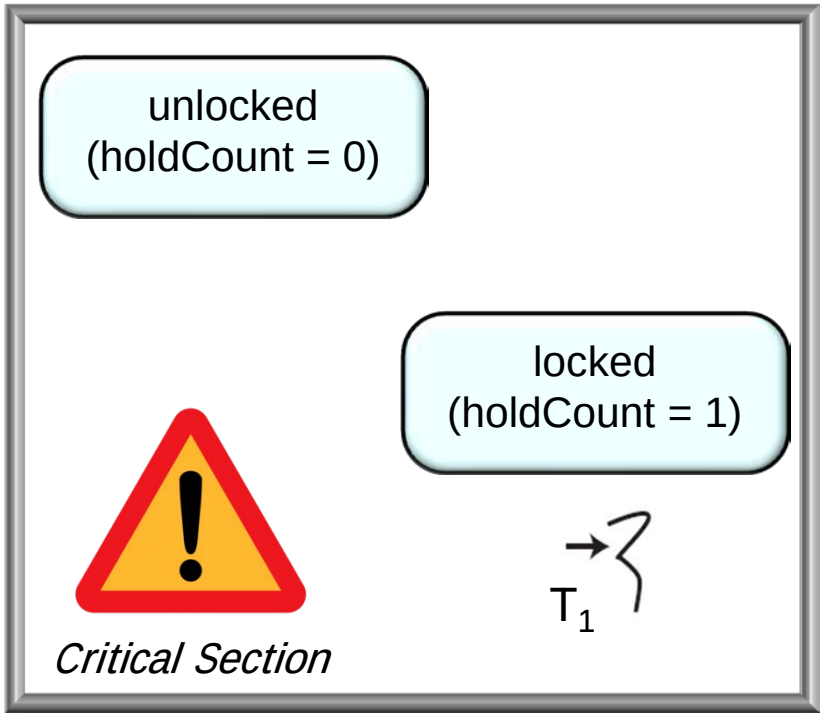
See mail.openjdk.java.net/pipermail/core-libs-dev/2010-May/004165.html

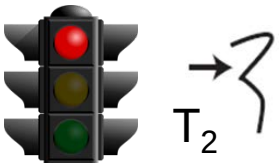
# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile

```
...
// Called by thread T2
String s = q.take();
...
```

**ArrayBlockingQueue**

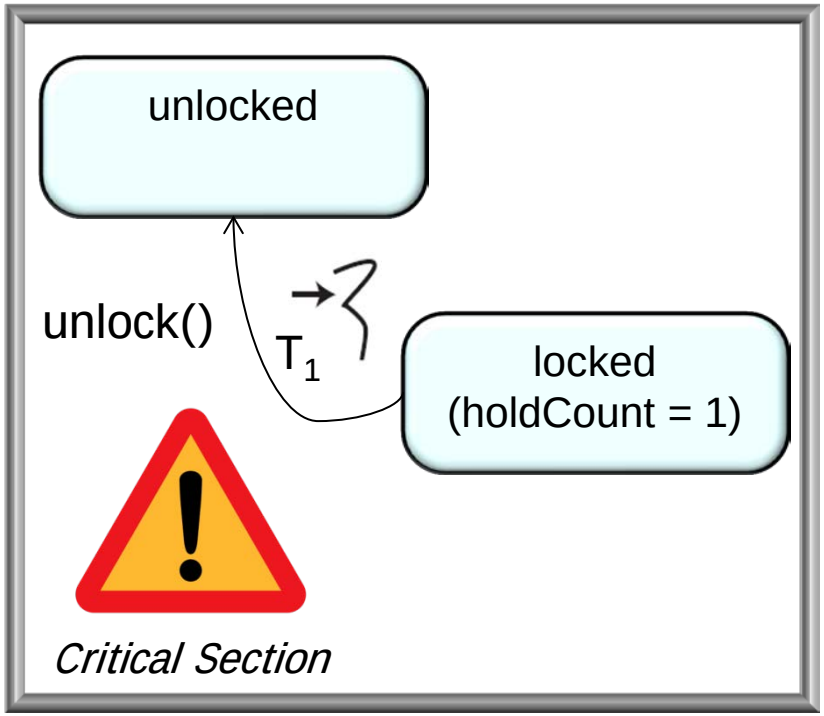


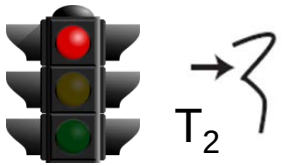
# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile

**ArrayBlockingQueue**



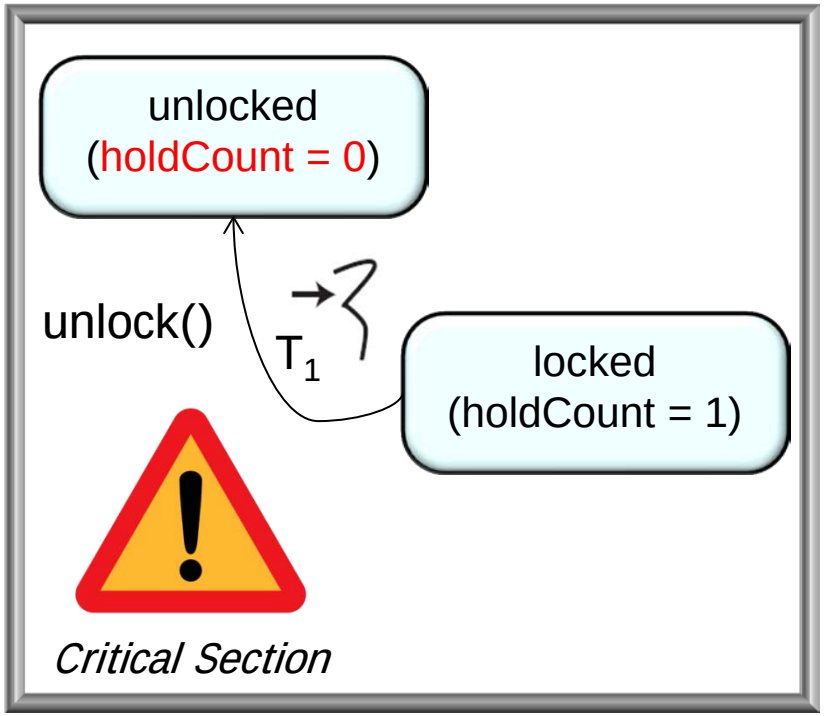

```
public class ArrayBlockingQueue<E>
          extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
...
public E take() ... {
  final ReentrantLock lock
    = this.lock;
  lock.lockInterruptibly();
  try { ...
  } finally {
    lock.unlock();
  }
  ...
```



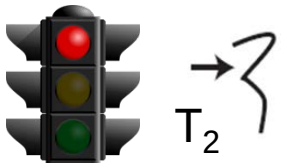
# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile

**ArrayBlockingQueue**



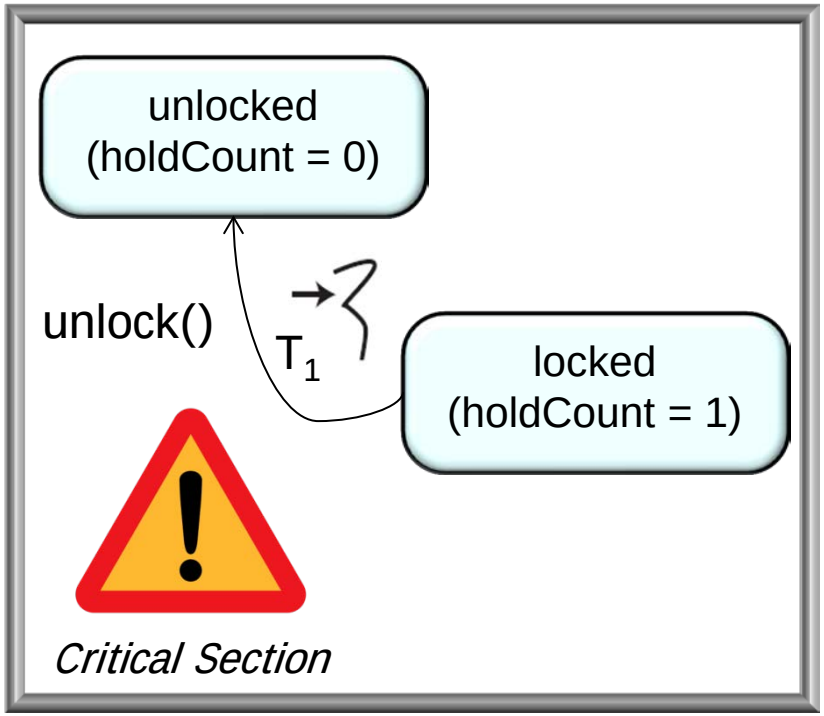

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
         implements BlockingQueue<E>,
                java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock
      = this.lock;
    lock.lockInterruptibly();
    try { ...
    } finally {
      lock.unlock();
    }
    ...
```



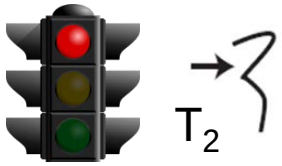
# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile

**ArrayBlockingQueue**




```
public class ArrayBlockingQueue<E>
          extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock
      = this.lock;
    lock.lockInterruptibly();
    try { ...
    } finally {
      lock.unlock();
    }
    ...
```

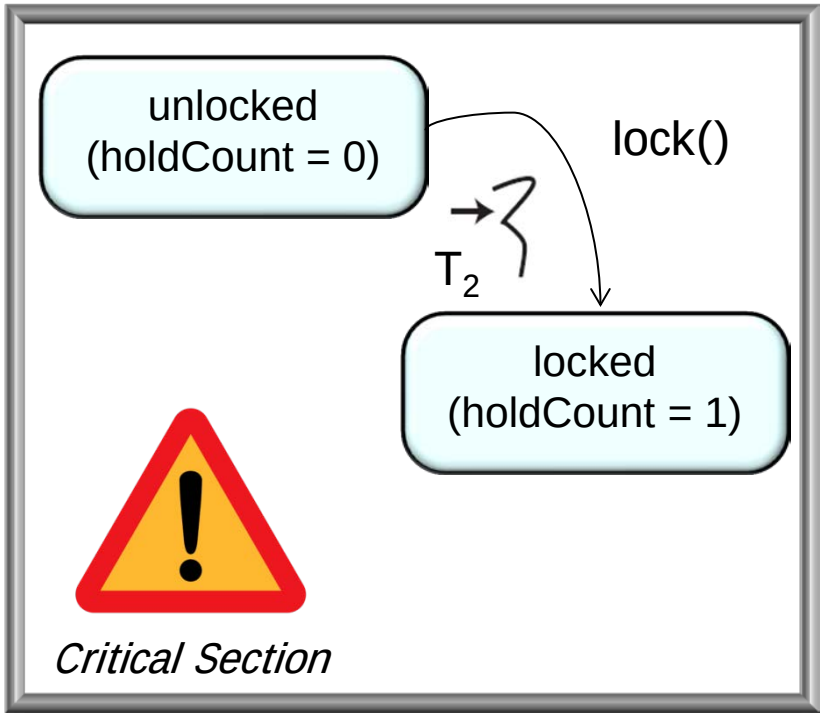



tutorials.jenkov.com/java-concurrency/locks.html#finally has more on finally

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile

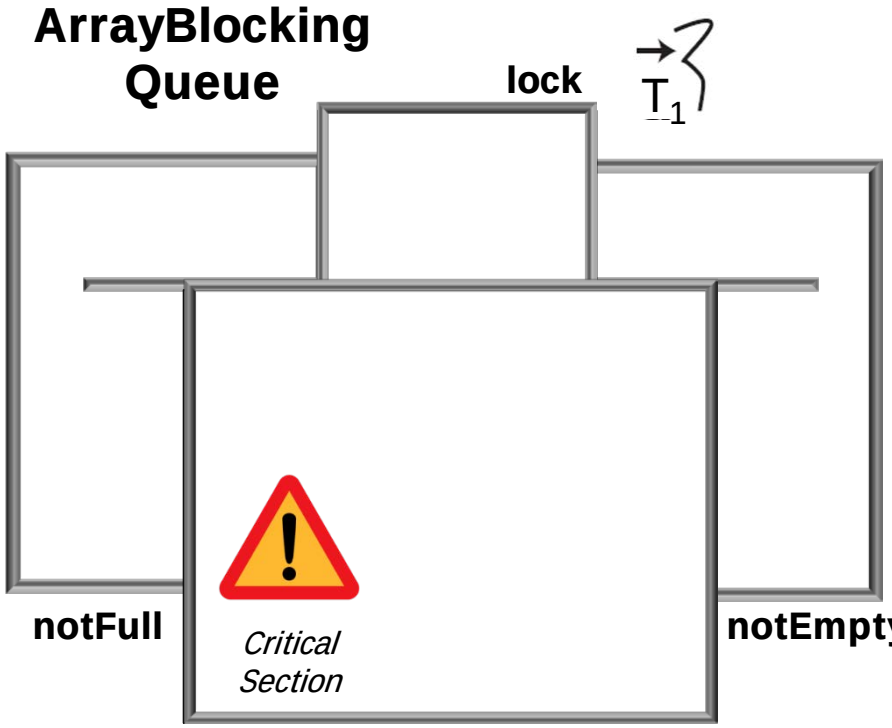
**ArrayBlockingQueue**




```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
         implements BlockingQueue<E>,
                java.io.Serializable {
...
public E take() ... {
  final ReentrantLock lock
    = this.lock;
  lock.lockInterruptibly();
  ...
```

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile

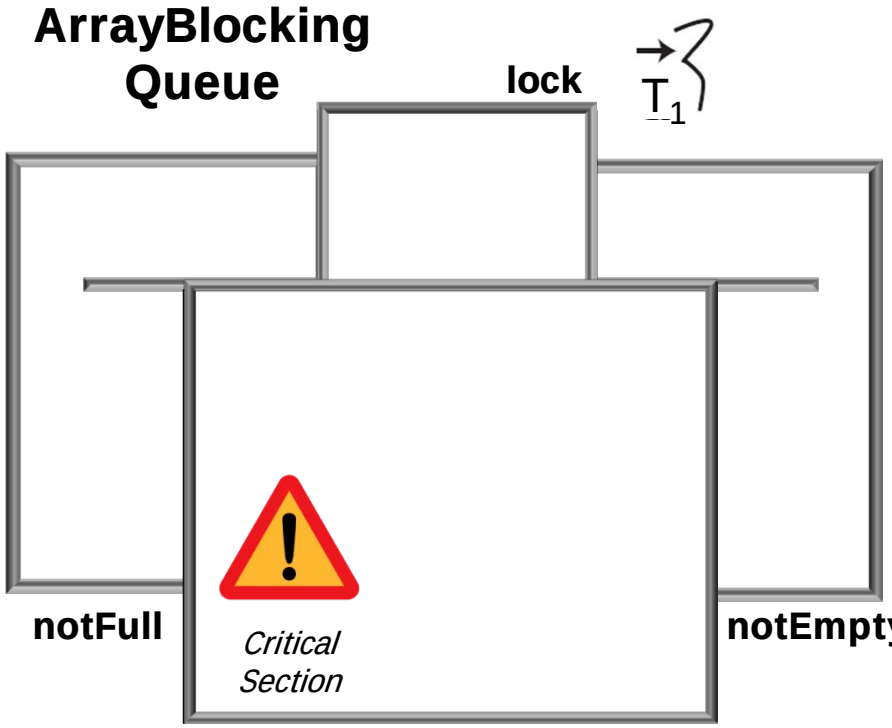



```
public class ArrayBlockingQueue<E>
          extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
...
public E take() ... {
  final ReentrantLock lock
    = this.lock;
  lock.lockInterruptibly();
  try {
    while (count == 0)
      notEmpty.await();
    return extract();
  } finally {
    lock.unlock();
  }
}
```

Upcoming part on "Java ConditionObject" has more on ArrayBlockingQueue

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile

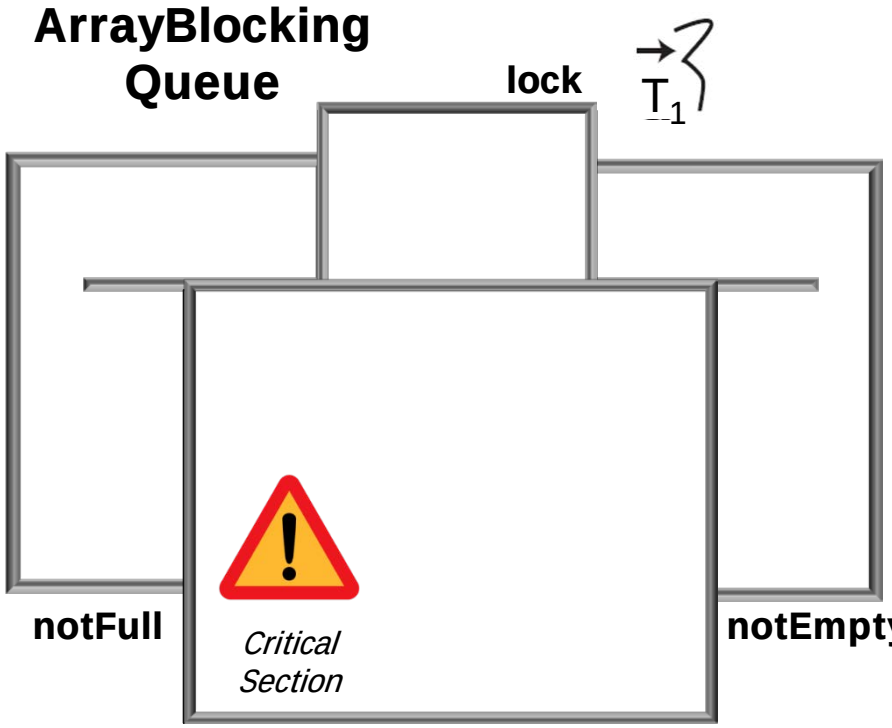



```
public class ArrayBlockingQueue<E>
          extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
...
public E take() ... {
  final ReentrantLock lock
    = this.lock;
  lock.lockInterruptibly();
  try {
    while (count == 0)
      notEmpty.await();
    return extract();
  } finally {
    lock.unlock();
  }
}
```

www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf presents *Monitor Object*

# Using ReentrantLock in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - Data members needn't be defined as volatile



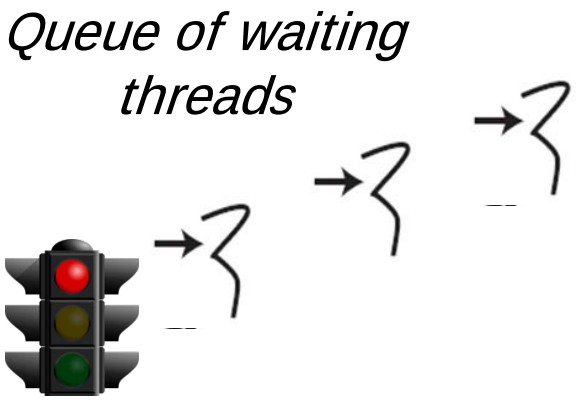

```
public class ArrayBlockingQueue<E>
        extends AbstractQueue<E>
    implements BlockingQueue<E>,
            java.io.Serializable {
...
public E take() ... {
  final ReentrantLock lock
    = this.lock;
  lock.lockInterruptibly();
  try {
    while (count == 0)
      notEmpty.await();
    return extract();
  } finally {
    lock.unlock();
  }
}
```

See upcoming parts on "the *Monitor Object* pattern"

# Summary

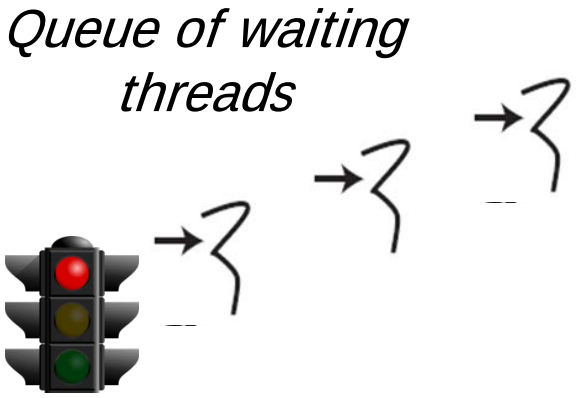
# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism



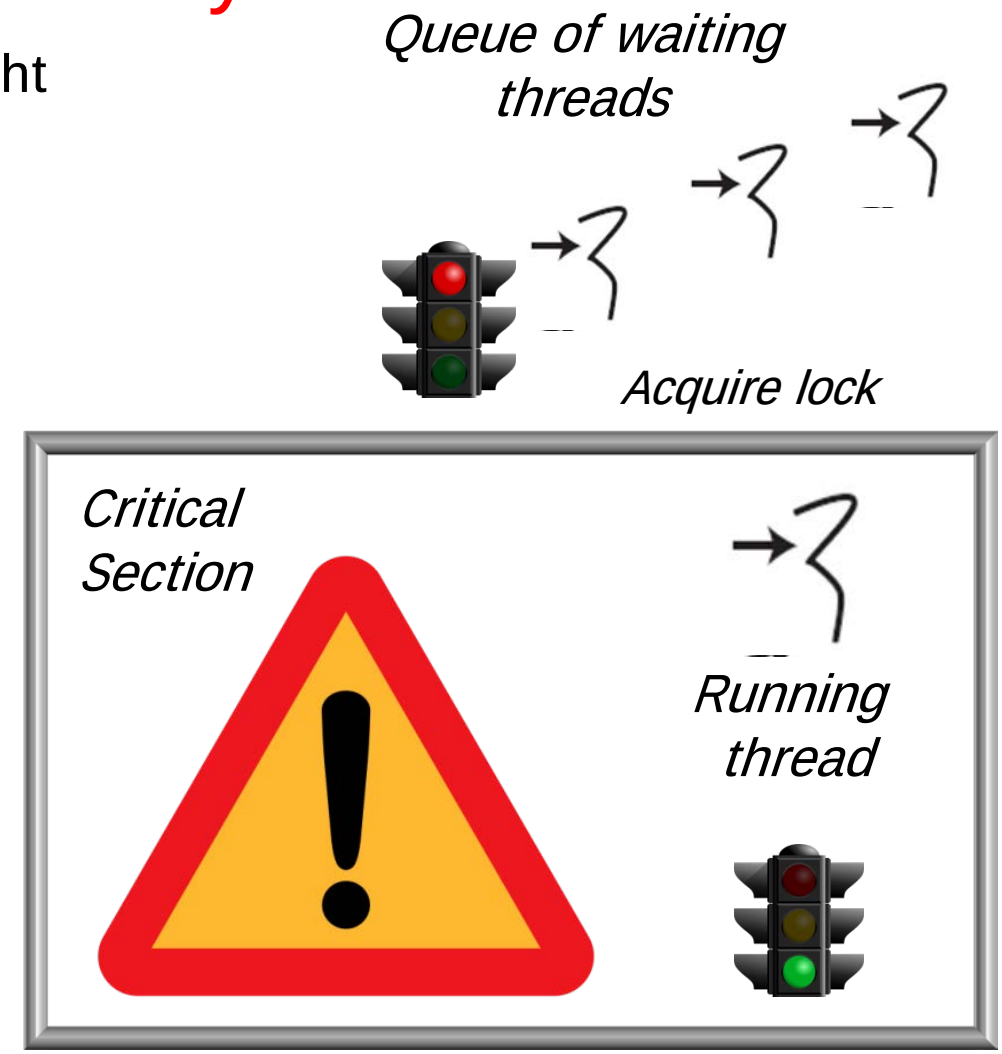
# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism




# Summary

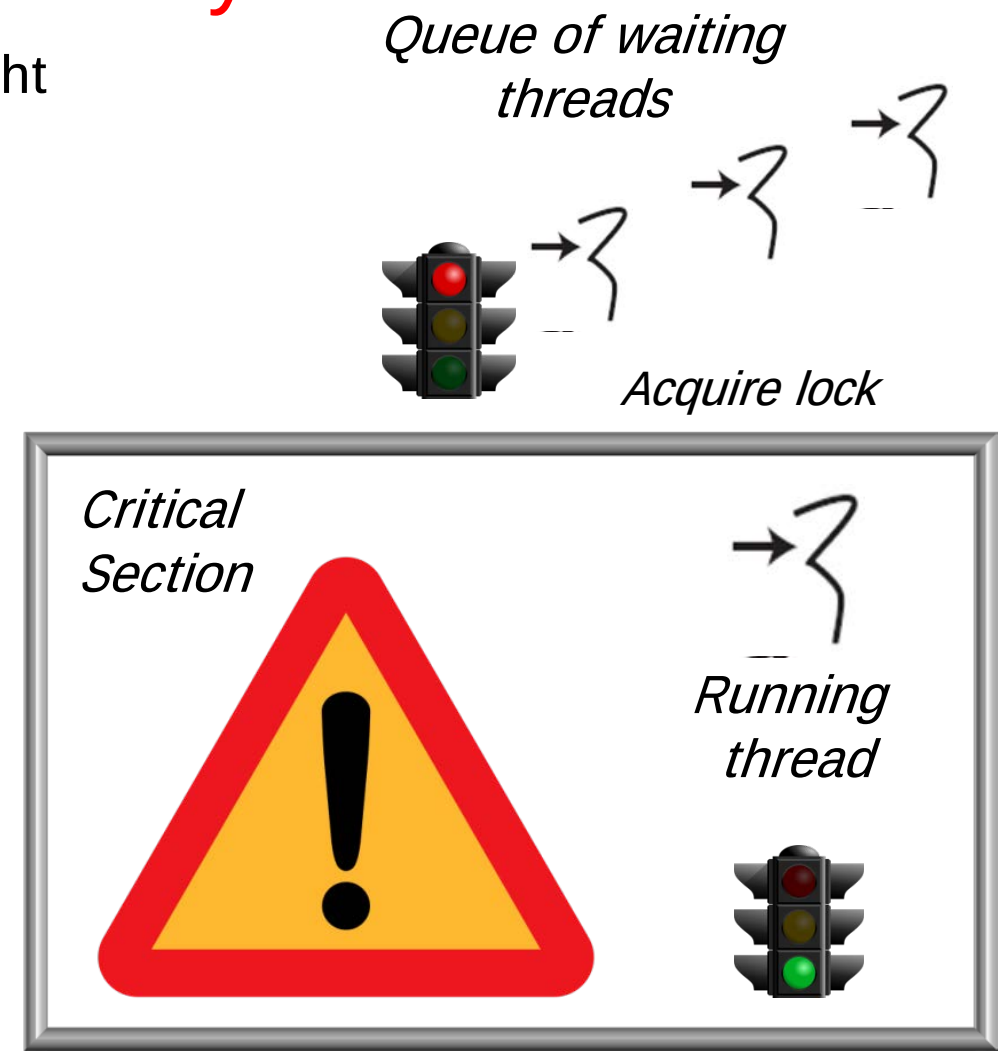
- ReentrantLock provides a lightweight mutual exclusion mechanism



# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism
  - Similar to built-in monitor lock used in Java's synchronized methods & statements

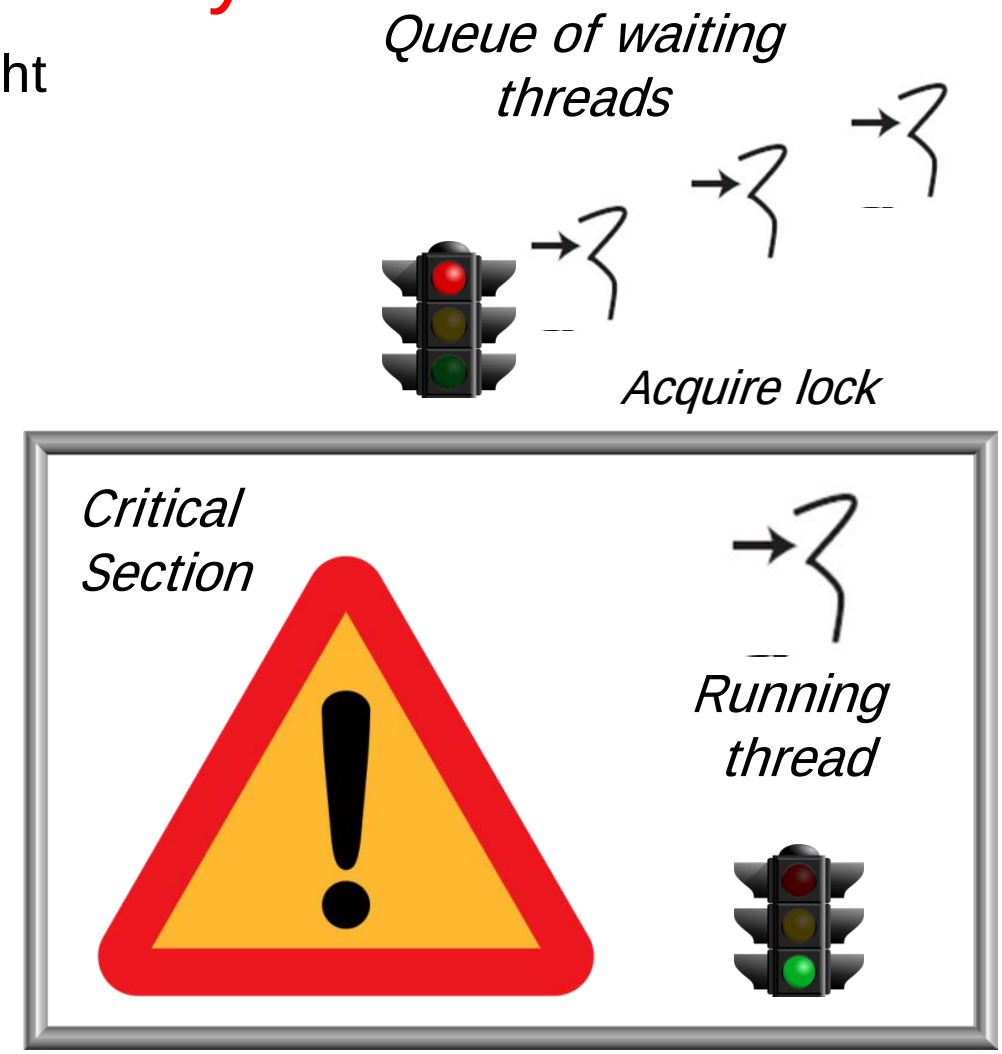
| | |
|---|---|
| void | lock()<br>Acquires the lock. |
| void | unlock()<br>Attempts to release this lock. |



# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism
  - Similar to built-in monitor lock used in Java's synchronized methods & statements
    - But has extended capabilities

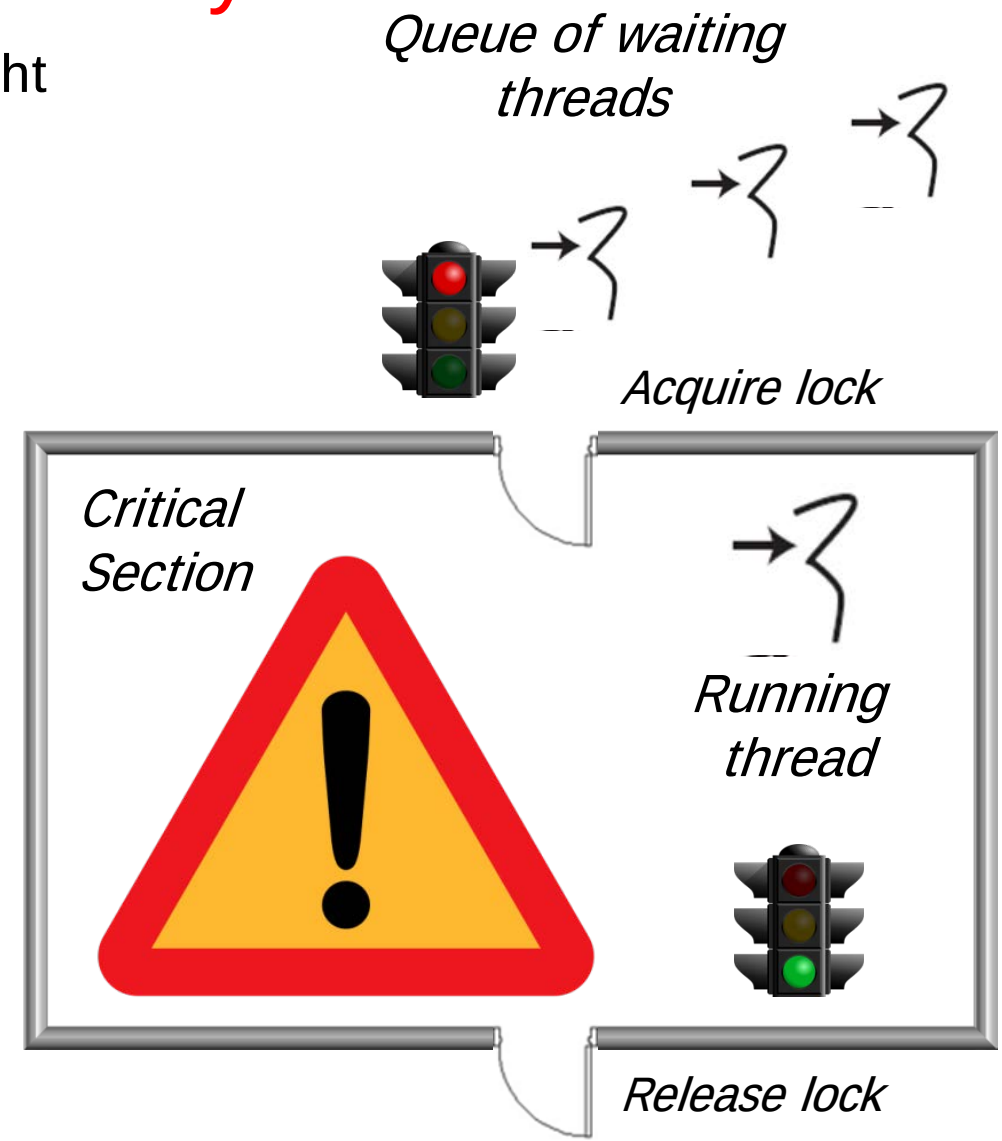
| | |
|---|---|
| boolean | tryLock()<br>Acquires the lock only if it is not held by another thread at the time of invocation. |
| void | lockInterruptibly()<br>Acquires the lock unless the current thread is `interrupted`. |
| boolean | tryLock(long timeout, TimeUnit unit)<br>Acquires the lock if it is not held by another thread within the given waiting time and the current thread has not been `interrupted`. |



# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism
  - It provides more capabilities that built-in Java monitor locks
  - It must be used via a “fully bracketed” protocol

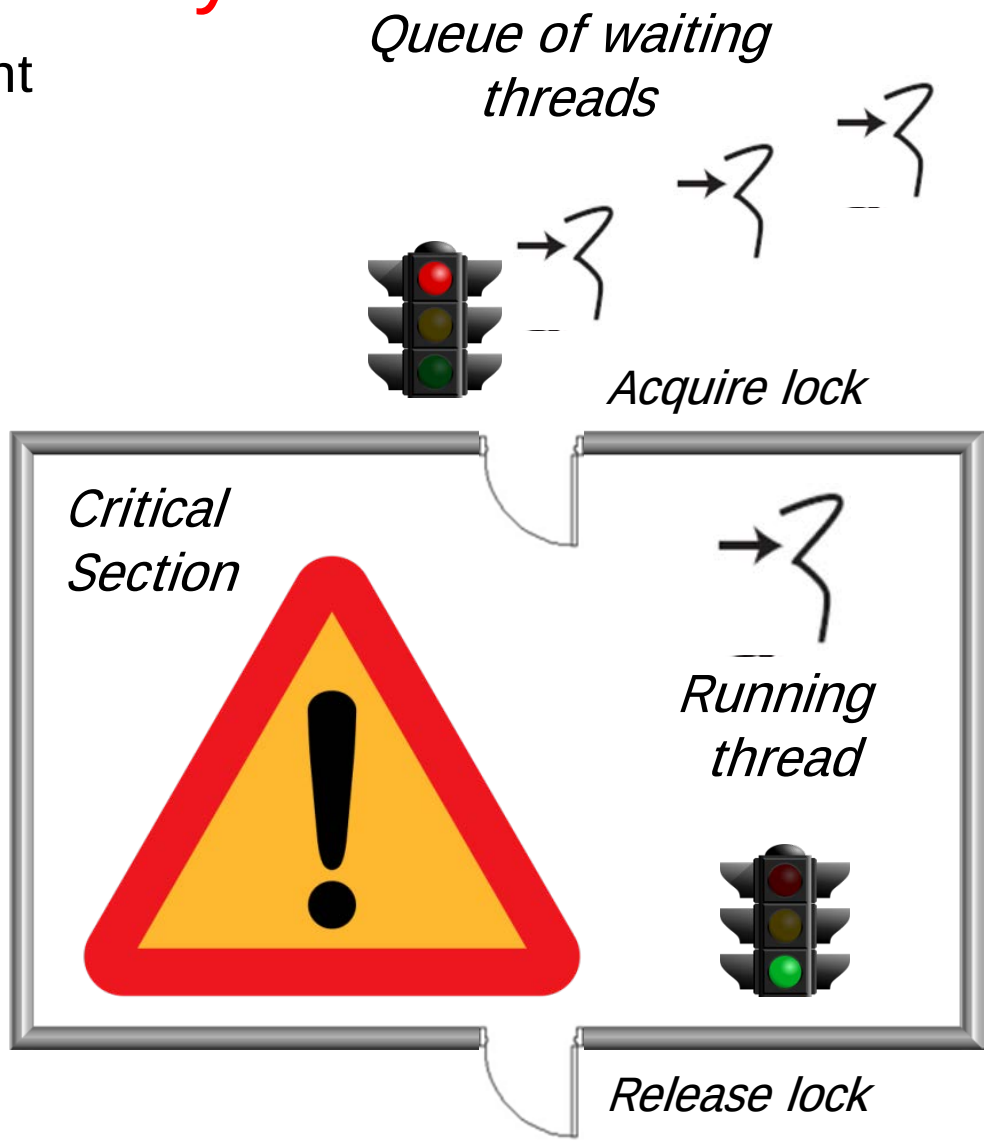
```
{
  ReentrantLock lock
    = this.lock;
  lock.lock();
  try { ...
  } finally {
    lock.unlock();
  }
}
```



# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism
  - It provides more capabilities that built-in Java monitor locks
  - It must be used via a "fully bracketed" protocol

```
void write_to_file
   (std::ofstream &file,
    const std::string &msg)
{
  static std::mutex mutex;

  std::lock_guard<std::mutex>
    lock(mutex);

  file << msg << std::endl;
}
```

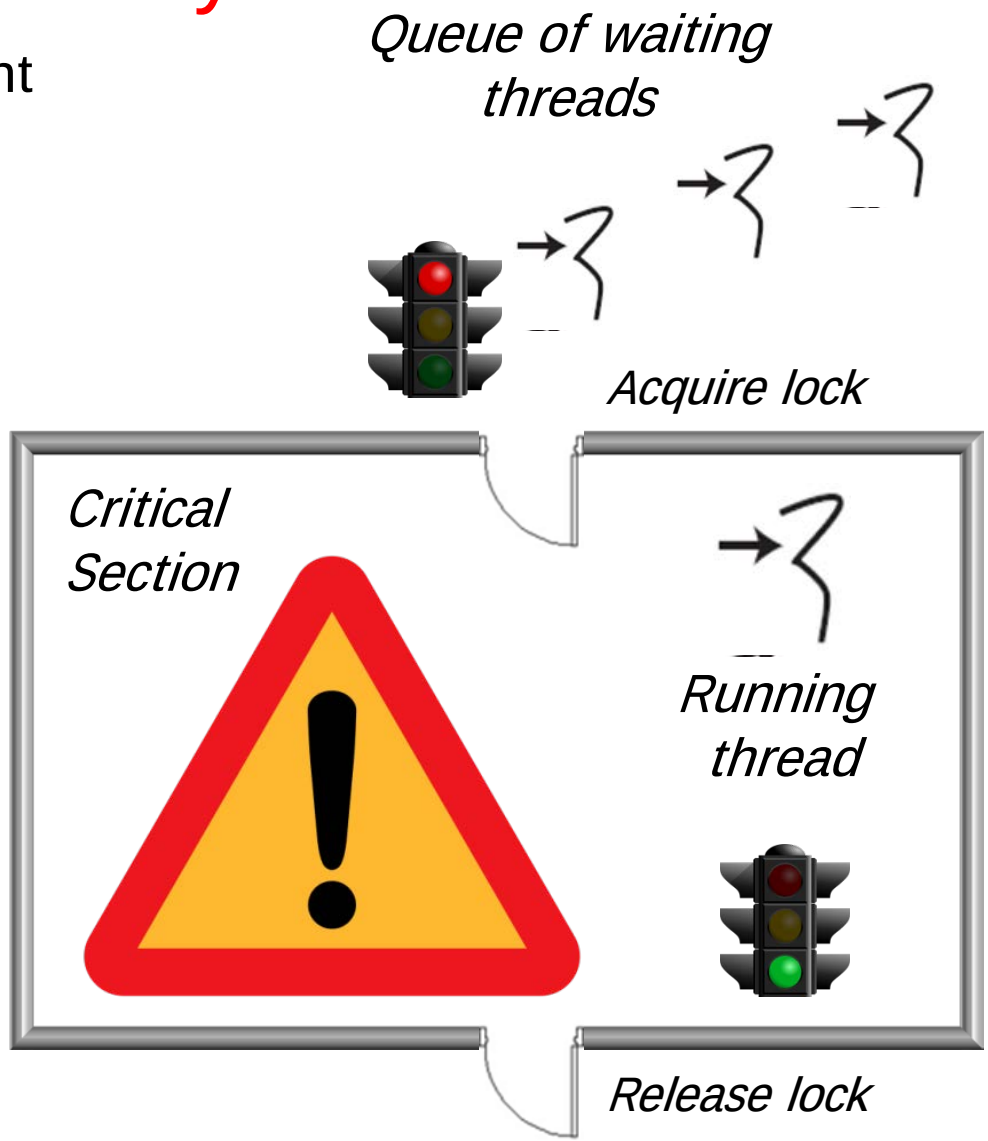



www.dre.vanderbilt.edu/~schmidt/PDF/locking-patterns.pdf has more info

# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism
  - It provides more capabilities that built-in Java monitor locks
  - It must be used via a "fully bracketed" protocol

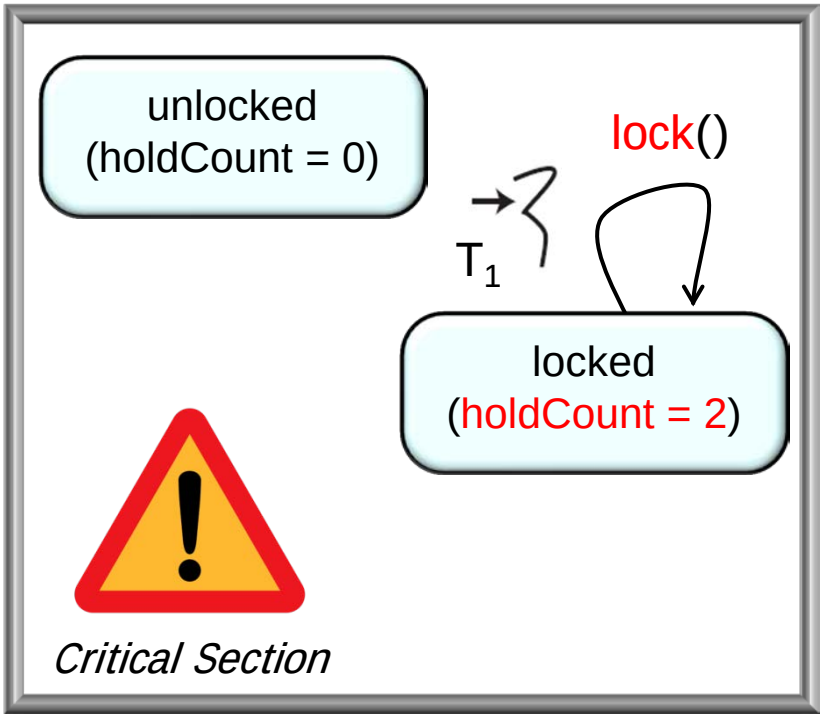
```
void write_to_file
   (std::ofstream &file,
    const std::string &msg)
{
  static std::mutex mutex;

  std::lock_guard<std::mutex>
    lock(mutex);

  file << msg << std::endl;
}
```




en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization has more info

# Summary

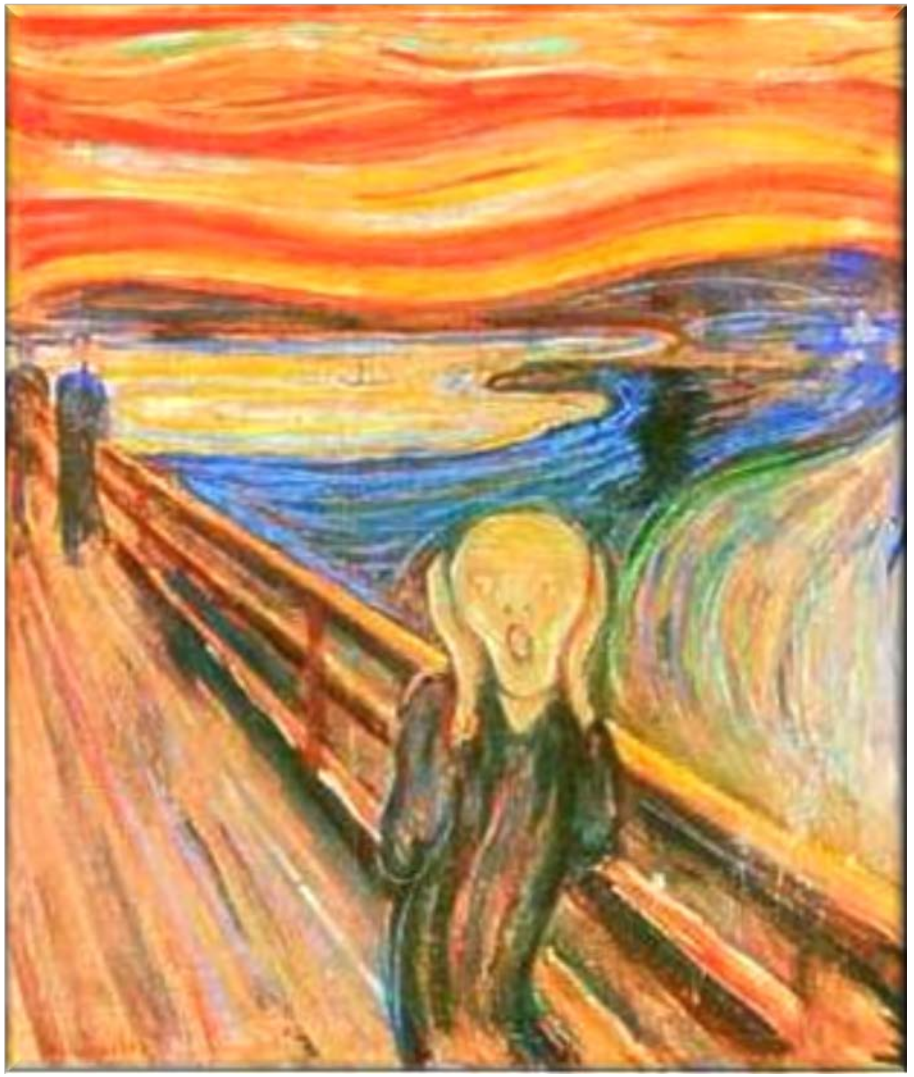
- ReentrantLock provides a lightweight mutual exclusion mechanism
  - It provides more capabilities that built-in Java monitor locks
  - It must be used in a “fully bracketed” manner
  - A ReentrantLock supports recursive lock semantics




en.wikipedia.org/wiki/Reentrant_mutex has more info on recursive locks

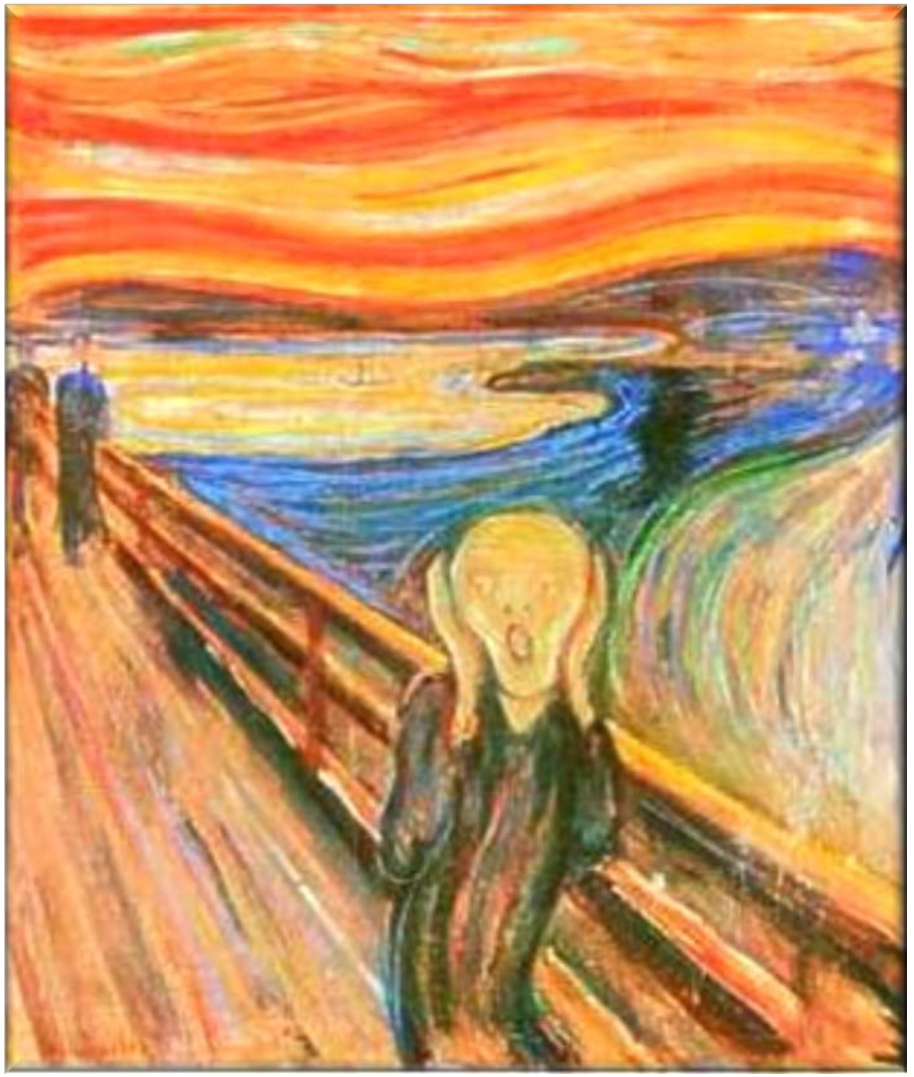
# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism
- ReentrantLocks can be tedious & error-prone to program due to common mistakes



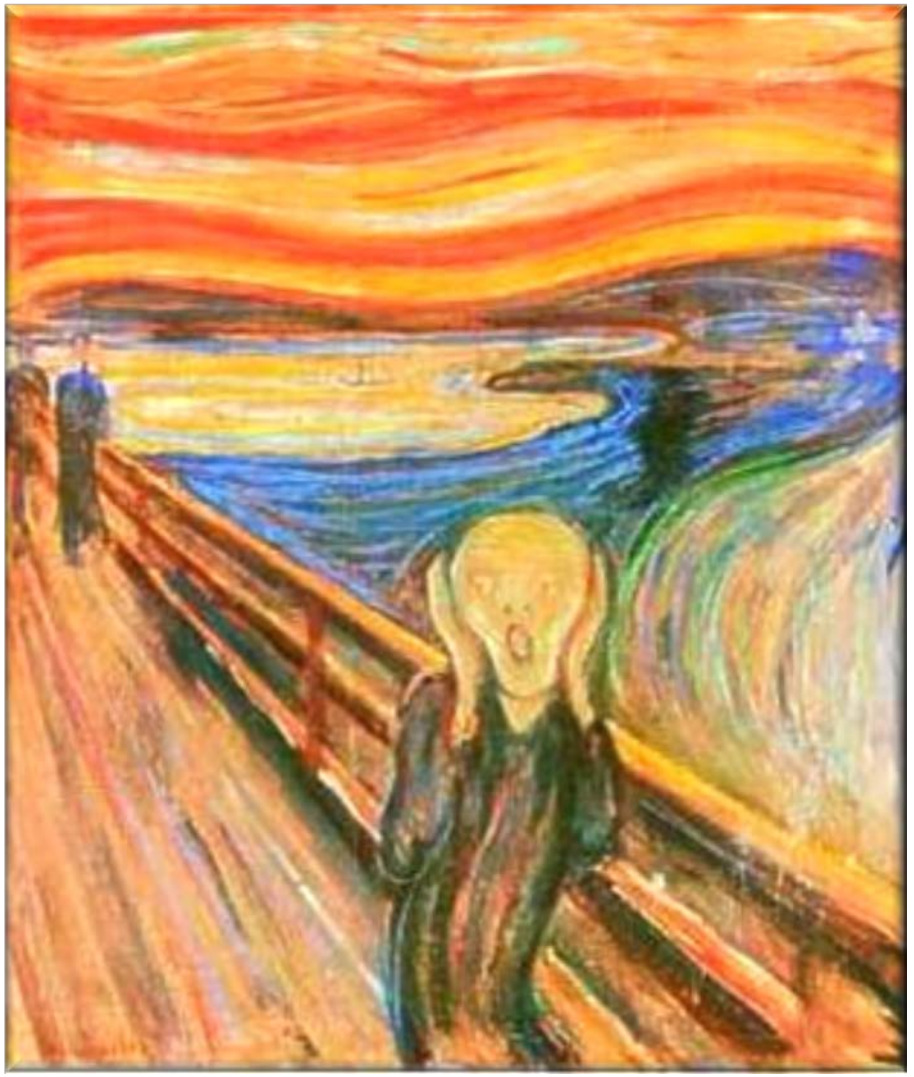
# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism
- ReentrantLocks can be tedious & error-prone to program due to common mistakes, e.g.
  - Acquiring a lock & forgetting to release it



# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism
- ReentrantLocks can be tedious & error-prone to program due to common mistakes, e.g.
  - Acquiring a lock & forgetting to release it
  - Releasing a lock that was never requested

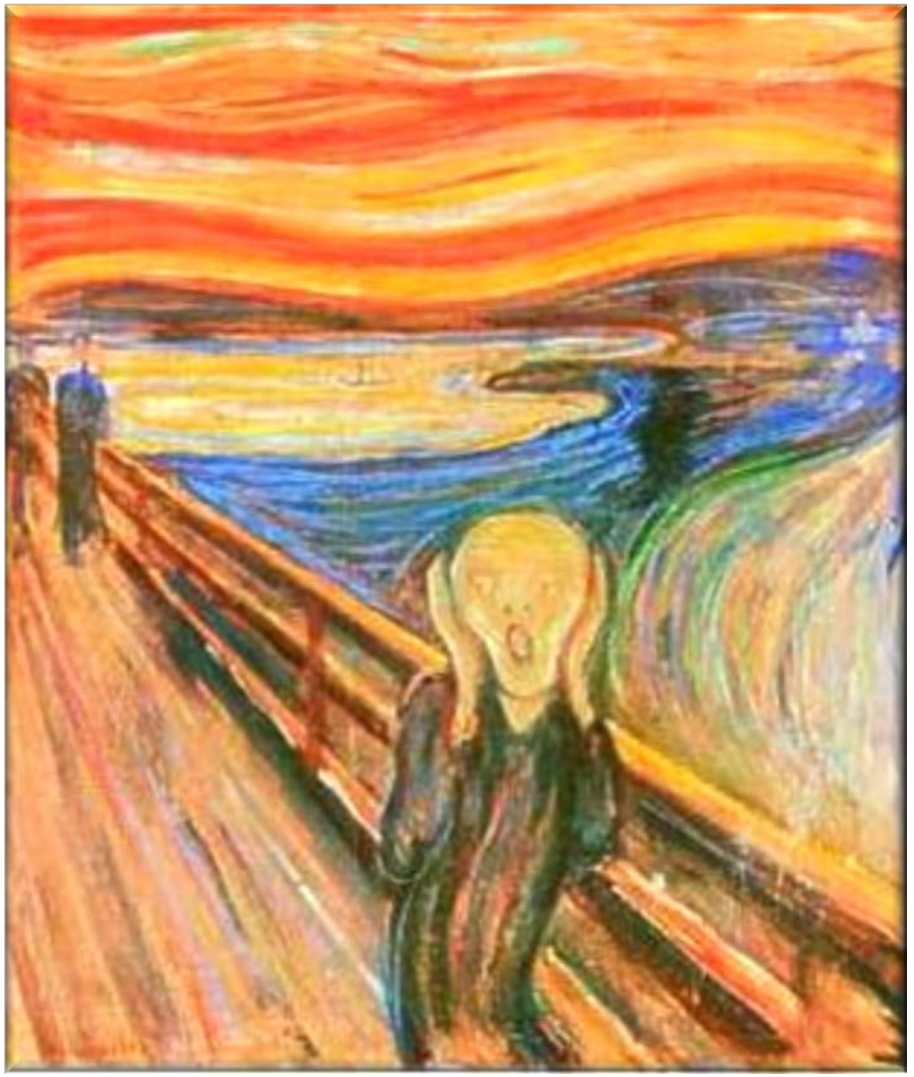


# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism
- ReentrantLocks can be tedious & error-prone to program due to common mistakes, e.g.
  - Acquiring a lock & forgetting to release it
  - Releasing a lock that was never requested
  - Holding a lock for a long time without needing it

# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism
- ReentrantLocks can be tedious & error-prone to program due to common mistakes, e.g.
  - Acquiring a lock & forgetting to release it
  - Releasing a lock that was never requested
  - Holding a lock for a long time without needing it
  - Accessing a resource without acquiring a lock for it first (or after releasing it)



See upcoming parts on “Java Built-in Monitor Objects”

# Summary

- ReentrantLock provides a lightweight mutual exclusion mechanism
- ReentrantLocks can be tedious & error-prone to program due to common mistakes
- ReentrantLock is used throughout Android
  - e.g., java.util.concurrent & java.util.concurrent.lock

package **Added in API level 1**

## java.util.concurrent.locks

Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors. The framework permits much greater flexibility in the use of locks and conditions, at the expense of more awkward syntax.
The `Lock` interface supports locking disciplines that differ in semantics (reentrant, fair, etc), and that can be used in non-block-structured contexts including hand-over-hand and lock reordering algorithms. The main implementation is `ReentrantLock`.

package **Added in API level 1**

## java.util.concurrent

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the `java.util.concurrent.locks` and `java.util.concurrent.atomic` packages.

libcore/luni/src/main/java/java/util/concurrent has the source code