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

See [en.wikipedia.org/wiki/Race_condition](en.wikipedia.org/wiki/Race_condition)
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 via 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
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 via 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 via 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 via 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](en.wikipedia.org/wiki/Linearizability)
A mutex is typically implemented in hardware via atomic operations.

Implemented in Java via the compareAndSwap*() methods in the Unsafe class.

Overview of Mutual Exclusion Locks

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 Variables & 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.
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

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

<<Java Class>>

ReentrantLock

ReentrantLock()
ReentrantLock(boolean)
lock():void
lockInterruptibly():void
tryLock():boolean
tryLock(long, TimeUnit):boolean
unlock():void
newCondition():Condition
gHoldCount():int
isHeldByCurrentThread():boolean
isLocked():boolean
isFair():boolean
hasQueuedThreads():boolean
hasQueuedThread(Thread):boolean
getQueueLength():int
hasWaiters(Condition):boolean
gWaitQueueLength(Condition):int
toString()
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](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html)
Overview of ReentrantLock

• Provide mutual exclusion to concurrent Java programs
• Implants 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
Overview of ReentrantLock

- 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
Overview of ReentrantLock

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

```java
class ReentrantLock implements Lock, java.io.Serializable {
    final Sync sync;

    /** Performs sync mechanics */
    final Sync sync;
    ...
```
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 {

    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
- 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();
    }

    ...
```
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();
    }
    ...
```
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();
    }

    ...}
```
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();
    }

    ...
```
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();
    }

    ...
```

13
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();
    }

    ...

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

In contrast, Java’s synchronized methods/blocks 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>Method Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>lock() – Acquires the lock</td>
</tr>
<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>
Overview of ReentrantLock

- A ReentrantLock supports “recursive mutex” semantics
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 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 behavior 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
class ReentrantLock
    implements Lock,
    java.io.Serializable {
    ...
    public void lock() {
        sync.lock();
    }
    ...
}
```

See [en.wikipedia.org/wiki/Spinlock](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 earlier part 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
public 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

```java
public class ReentrantLock extends 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
    - Exception 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
There are many other ReentrantLock methods

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

These methods go above & beyond what’s available from Java’s synchronized statements/methods
There are many other ReentrantLock methods

<table>
<thead>
<tr>
<th>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</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean isFair() – Returns true if this lock has fairness set true</td>
</tr>
<tr>
<td>boolean isLocked() – Queries if this lock is held by any thread</td>
</tr>
<tr>
<td>Condition newCondition() – 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

- There are many other ReentrantLock methods

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

Not very useful due to non-determinism of concurrency.
### 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 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)
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
- Know how to use ReentrantLock in Java programs

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

A bounded blocking queue backed by an array. This queue orders elements FIFO (first-in-first-out). The head of the queue is the element that has been on the queue the longest time. The tail of the queue is the 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 Java
Using ReentrantLock in Java

- **ArrayBlockingQueue** is a blocking bounded FIFO queue

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

Class **ArrayBlockingQueue**<E>

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

**Type Parameters:**

- `E` - the type of elements held in this collection

**All Implemented Interfaces:**

- Serializable, Iterable<E>, Collection<E>, BlockingQueue<E>, Queue<E>

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

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.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html)
Using ReentrantLock in Java

- ArrayBlockingQueue is a blocking bounded FIFO queue

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

Class AbstractQueue<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractQueue<E>

Type Parameters:
    E - the type of elements held in this collection

All Implemented Interfaces:
    Iterable<E>, Collection<E>, Queue<E>

Direct Known Subclasses:
    ArrayBlockingQueue, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue,
    PriorityBlockingQueue, PriorityQueue, SynchronousQueue

public abstract class AbstractQueue<E>
    extends AbstractCollection<E>
    implements Queue<E>

This class provides skeletal implementations of some Queue operations. The implementations in this class are appropriate when the base implementation does not allow null elements. Methods add, remove, and element are based on offer, poll, and peek, respectively, but throw exceptions instead of indicating failure via false or null returns.

See docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html
```
Using ReentrantLock in Java

- ArrayBlockingQueue is a blocking bounded FIFO queue

```java
public class ArrayBlockingQueue&lt;E&gt; extends AbstractQueue&lt;E&gt; implements BlockingQueue&lt;E&gt;, java.io.Serializable {
```

**Interface BlockingQueue&lt;E&gt;**

**Type Parameters:**

- `E` - the type of elements held in this collection

**All Superinterfaces:**

- `Collection&lt;E&gt;`, `Iterable&lt;E&gt;`, `Queue&lt;E&gt;`

**All Known Subinterfaces:**

- `BlockingDeque&lt;E&gt;`, `TransferQueue&lt;E&gt;`

**All Known Implementing Classes:**

- `ArrayBlockingQueue`, `DelayQueue`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, `LinkedTransferQueue`, `PriorityBlockingQueue`, `SynchronousQueue`

```java
public interface BlockingQueue&lt;E&gt;
extends Queue&lt;E&gt;
```

A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)
Using ReentrantLock in Java

- ArrayBlockingQueue is a blocking bounded FIFO queue

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

    ...

We’ll consider both the interface & implementation of ArrayBlockingQueue
Using ReentrantLock in Java

- 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;

    See www.dre.vanderbilt.edu/~schmidt/C++2java.html
Using ReentrantLock in Java

- ArrayBlockingQueue is a blocking bounded FIFO queue

```java
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 Java

• ArrayBlockingQueue is a blocking bounded FIFO queue

```java
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;
}
```

Object state being protected by the lock
ArrayBlockingQueue is a blocking bounded FIFO queue.

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

    private final ReentrantLock lock;

    private final Object[][] items;

    private int takeIndex;
    private int putIndex;

    private int count;

    // 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;
```
ArrayBlockingQueue is a blocking bounded FIFO queue.

```java
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;
```

Fields needn't be defined as volatile objects since ReentrantLock handles all atomicity & visibility issues.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html
ArrayBlockingQueue is a blocking bounded FIFO queue

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

Critical Section

unlocked (holdCount = 0)
Using ReentrantLock in Java

- ArrayBlockingQueue is a blocking bounded FIFO queue

```java
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 Java

- ArrayBlockingQueue is a blocking bounded FIFO queue

```java
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 Java

- ArrayBlockingQueue is a blocking bounded FIFO queue

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

**ArrayBlockingQueue**

- Unlocked (holdCount = 0)
- Locked (holdCount = 1)

Critical Section

A call to take() from another thread T2 will block until thread T1 is finished
Using ReentrantLock in Java

- ArrayBlockingQueue is a blocking bounded FIFO queue

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();
        }
    }
    ...

When thread \( T_1 \) is finished in take() it unlocks the lock
Using ReentrantLock in Java

- ArrayBlockingQueue is a blocking bounded FIFO queue

```
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();
        }
    }

    ...
```

At this point holdCount reverts back to 0
ArrayBlockingQueue is a blocking bounded FIFO queue

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 Java

- ArrayBlockingQueue is a blocking bounded FIFO queue

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

Thread T2 can now enter the critical section of ArrayBlockingQueue
ArrayBlockingQueue is a blocking bounded FIFO queue

```java
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 lesson on “Java ConditionObject” has more coverage of ArrayBlockingQueue
ArrayBlockingQueue is a blocking bounded FIFO queue.

```java
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 [www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf)
End of Java ReentrantLock (Part 3)