Java ReentrantLock: Example Application

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
- Be aware of reentrant mutex semantics
- Know the key methods defined by the Java ReentrantLock class
- Master how to apply ReentrantLock in practice
Applying Reentrant Lock in Practice
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking FIFO queue

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

class ArrayBlockingQueue<E>

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>

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](docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html)
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking FIFO queue

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

See docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking FIFO queue

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

  // Constructor and methods...

  // Interface BlockingQueue<E>

  public interface BlockingQueue<E>
    extends Queue<E>

  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
```
Applying ReentrantLock in Practice

• ArrayBlockingQueue is a bounded blocking FIFO queue

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

...
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking 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;

    ReentrantLock used in lieu of Java’s built-in monitor objects due to their limitations

    See www.dre.vanderbilt.edu/~schmidt/C++2java.html#concurrency
```
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking 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;
```

Object state that’s being protected by the lock
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking 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 since ReentrantLock handles all of the atomicity, visibility, & ordering issues

    See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Lock.html
```
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking FIFO queue

```java
ArrayBlockingQueue<String> q = new ArrayBlockingQueue<>(10);
```

Create a bounded blocking queue that can store up to 10 items
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking FIFO queue

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

Critical Section

Thread T₁ acquires the lock & enters the critical section
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking 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();
    ...

    The lock’s hold count is incremented by 1
```

Critical Section
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking FIFO queue

```java
ArrayBlockingQueue<String> q = new ArrayBlockingQueue<>(10);
...
// Called by thread T2
String s = q.take();
...
```

A call to `take()` from thread $T_2$ will block until thread $T_1$ is finished.
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking 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 T₁ finishes in take() it unlocks the lock
```

Critical Section

![Diagram showing the lifecycle of a lock in ArrayBlockingQueue.]
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking 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

    locked
    (holdCount = 1)

    unlocked
    (holdCount = 0)

    Critical Section

    T1

    T2

    ArrayBlockingQueue
```
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking 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();
    }
    ...
  }

  Ensure lock is always released when T₁ exits the critical section
```

See tutorials.jenkov.com/java-concurrency/locks.html#finally
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking 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();
        ...

        Thread T2 can now enter the critical section of take() & start running
    }
```

ArrayBlockingQueue

- unlocked (holdCount = 0)
- locked (holdCount = 1)

Critical Section
ArrayBlockingQueue needs to use more than ReentrantLock to implement its semantics.

```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();
    }

    A Java ConditionObject is used to coordinate multiple threads
```

Upcoming lesson on "Java ConditionObject" shows more on ArrayBlockingQueue
ArrayBlockingQueue needs to use more than ReentrantLock to implement its semantics.

```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();
    }
  }

  Critical Section

  notFull
  lock
  T1

  notEmpty


These mechanisms implement Guarded Suspension & Monitor Object patterns

YOU SHALL NOT PASS
```
End of Java ReentrantLock: Example Application