Example Application of 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 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
  • i.e., in the context of the Java ArrayBlockingQueue class

```
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.

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.
```
Applying Reentrant Lock in Practice
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 {
```

**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>

```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](docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.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 {
...
```

See [docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html](https://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 {
```

**Interface BlockingQueue<E>**

**Type Parameters:**

- E - the type of elements held in this collection

**All Superinterfaces:**

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

**All Known Subinterfaces:**

- BlockingDeque<E>, TransferQueue<E>

**All Known Implementing Classes:**

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

```java
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

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

We’ll consider both the interface & implementation of 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 {

    ...  
    // 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](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;
```

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

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();
...
```

Thread $T_1$ 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
```
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 T2 will block until thread T1 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 $T_1$ finishes in `take()` it unlocks the lock

Diagram:
- ArrayBlockingQueue
- Critical Section
- Unlock
- Locked (holdCount = 1)
- Unlocked (holdCount = 0)
- $T_1$ finishing in `take()` unlocks the lock
- $T_2$
Applying ReentrantLock in Practice

- ArrayBlockingQueue is a bounded blocking 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 {
            ...}
        finally {
            lock.unlock();
        }
    }
```

At this point, holdCount reverts back to 0
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

```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 take() & start running
```

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

```
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.
Applying ReentrantLock in Practice

• 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();  
        }
    }

    These mechanisms implement Guarded Suspension & Monitor Object patterns

See en.wikipedia.org/wiki/Guarded_suspension &
www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf
End of Example Application of Java ReentrantLock