Java ReentrantLock
(Part 3)

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
- Know how to use ReentrantLock in Java programs
Using Reentrant Lock 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

```
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](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<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](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

```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
```
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
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
```
ArrayBlockingQueue is a blocking bounded FIFO queue

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

Thread $T_1$ acquires the lock & enters the critical section
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();
        ...
    }

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

Critical Section

<table>
<thead>
<tr>
<th>unlocked (holdCount = 0)</th>
<th>lock()</th>
<th>locked (holdCount = 1)</th>
</tr>
</thead>
</table>

T₁
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 thread T₂ will block until thread T₁ is finished
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

```
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

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

    ...
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 take() & start running
    }
    ...
```
ArrayBlockingQueue needs to use more than ReentrantLock to implement its semantics.

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

A Java ConditionObject is used to coordinate multiple threads.

Upcoming lesson on "Java ConditionObject" shows more on ArrayBlockingQueue.
Using ReentrantLock in Java

- 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

End of Java ReentrantLock (Part 3)
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
- Appreciate Java ReentrantLock usage considerations
ReentrantLock Usage Considerations
ReentrantLock Usage Considerations

- ReentrantLock must be used via a “fully bracketed” protocol

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

The thread that acquires the lock **must** be the one to release it
ReentrantLock Usage Considerations

- ReentrantLock must be used via a “fully bracketed” protocol
- This design is known as the “Scoped Locking” pattern

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

The finally clause ensures that the lock is released on all paths out the try clause.

```
begin ## Enter the critical section.
## Acquire the lock automatically.
## Execute the critical section.
do_something ();
## Release the lock automatically.
end ## Leave the critical section.
```

See [www.dre.vanderbilt.edu/~schmidt/PDF/locking-patterns.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/locking-patterns.pdf)
ReentrantLock Usage Considerations

- ReentrantLock must be used via a “fully bracketed” protocol
- This design is known as the “Scoped Locking” pattern
- Implemented implicitly via Java synchronized methods & statements

```java
void someMethod() {
    synchronized (this) {
        ...
    }
}
```

See lesson on “Java Built-in Monitor Object”
ReentrantLock Usage Considerations

- ReentrantLock must be used via a “fully bracketed” protocol
  - This design is known as the “Scoped Locking” pattern
  - Implemented implicitly via Java synchronized methods & statements
  - This pattern is commonly used in C++ (& C#) via constructors & destructors

```cpp
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;
}
```

See [en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization](en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization)
ReentrantLock Usage Considerations

- ReentrantLock supports “recursive mutex” semantics where a lock may be acquired multiple times by the same thread, without causing self-deadlock.

See en.wikipedia.org/wiki/Reentrant_mutex
ReentrantLock Usage Considerations

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

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

`onTick()` is called with the `mLock` held

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

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex24](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex24)
ReentrantLock Usage Considerations

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

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

```java
if (...) cancel();
```

---

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex24](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex24)
ReentrantLock Usage Considerations

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

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

`cancel()` also acquires `mLock`, so it must be recursive or self-deadlock will occur

```java
if (...) {
    cancel();
}
```

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex24](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex24)
ReentrantLock Usage Considerations

- ReentrantLocks can be tedious & error-prone to program due to common traps & pitfalls
• ReentrantLocks can be tedious & error-prone to program due to common traps & pitfalls, e.g.
• Holding a lock for a long time without needing it

```java
void someMethod() {
    ReentrantLock lock = this.lock;
    lock.lock();
    try {
        for (;;) {
            // Do something that doesn't involve lock
        }
    } finally {
        lock.unlock();
    }
}
```
ReentrantLock Usage Considerations

- ReentrantLocks can be tedious & error-prone to program due to common traps & pitfalls, e.g.

  - Holding a lock for a long time without needing it
  - Acquiring a lock & forgetting to release it

```java
void someMethod() {
    ReentrantLock lock = this.lock;
    lock.lock();
    ... // Critical section
    return;
}
```
ReentrantLock Usage Considerations

- ReentrantLocks can be tedious & error-prone to program due to common traps & pitfalls, e.g.
  - Holding a lock for a long time without needing it
  - Acquiring a lock & forgetting to release it
  - Releasing a lock that was never acquired
    - or has already been released

```java
void someMethod() {
    ReentrantLock lock = this.lock;
    // lock.lock();
    try {
        ... // Critical section
    } finally {
        lock.unlock();
    }
}
```
ReentrantLock Usage Considerations

- ReentrantLocks can be tedious & error-prone to program due to common traps & pitfalls, e.g.
  - Holding a lock for a long time without needing it
  - Acquiring a lock & forgetting to release it
  - Releasing a lock that was never acquired
  - Accessing a resource without acquiring a lock for it first
    - or after releasing it

```java
void someMethod() {
    ReentrantLock lock = this.lock;
    // lock.lock();
    try {
        ... // Critical section
    } finally {
        // lock.unlock();
    }
}
```

Compare with lesson on “Java Built-in Monitor Objects”
End of Java ReentrantLock (Part 4)