Java ConditionObject: 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 what condition variables are
- Note a human known use of condition variables
- Know what pattern they implement
- Recognize common use cases where condition variables are applied
- Recognize the structure & functionality of Java ConditionObject
- Know the key methods defined by the Java ConditionObject class
- Master the use of Condition Objects in practice
Applying Java Condition Object in Practice
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](docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html)
Applying Java ConditionObject in Practice

- ArrayBlockingQueue is a blocking bounded FIFO queue

```
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
• ArrayBlockingQueue is a blocking bounded FIFO queue

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

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 Java ConditionObject in Practice

- ArrayBlockingQueue is a blocking bounded FIFO queue

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

We’ll focus on both the interface & implementation of ArrayBlockingQueue
ArrayBlockingQueue is a blocking bounded FIFO queue

- It’s implemented using a dynamically sized array

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

    ... 

    /** The queued items */
    final Object[] items;

    /** items index for next take,
        poll, peek or remove */
    int takeIndex;

    /** items index for next put,
        offer, or add */
    int putIndex;

    /** Number of elements in
        the queue */
    int count;
    ...
```
ArrayBlockingQueue is a blocking bounded FIFO queue
- It’s implemented using a dynamically sized array

Applying Java ConditionObject in Practice

Object state that (1) must be protected from race conditions & (2) is used to coordinate concurrent put() & take() calls

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

    ...  
    /** The queued items */
    final Object[] items;

    /** items index for next take, poll, peek or remove */
    int takeIndex;

    /** items index for next put, offer, or add */
    int putIndex;

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

    ...  
```
Applying Java ConditionObject in Practice

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - It’s implemented using a dynamically sized array
  - It has a ReentrantLock & two ConditionObjects

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

    final ReentrantLock lock;

    final Condition notEmpty;
    final Condition notFull;

    ...  

    /** Main lock guarding access */
    final ReentrantLock lock;

    /** Condition for waiting takes */
    private final Condition notEmpty;

    /** Condition for waiting puts */
    private final Condition notFull;

    ...  
```

Used to protect the object state from race conditions

See earlier lesson on “Java ReentrantLock: Example Application’”
Applying Java ConditionObject in Practice

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - It’s implemented using a dynamically sized array
  - It has a ReentrantLock & two ConditionObjects

```java
public class ArrayBlockingQueue<E> extends AbstractQueue<E>
    implements BlockingQueue<E>, java.io.Serializable {
    ...  
    /** Main lock guarding access */
    final ReentrantLock lock;
    
    /** Condition for waiting takes */
    private final Condition notEmpty;
    
    /** Condition for waiting puts */
    private final Condition notFull;
    ...

    Two ConditionObjects separate waiting consumers & producers, thus reducing redundant wakeups & checking
```

See stackoverflow.com/questions/18490636/condition-give-the-effect-of-having-multiple-wait-sets-per-object
• ArrayBlockingQueue is a blocking bounded FIFO queue
  • It’s implemented using a dynamically sized array
• It has a ReentrantLock & two ConditionObjects

```java
public class ArrayBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
    java.io.Serializable {
...
  public ArrayBlockingQueue
      (int capacity, 
       boolean fair) {
    items =
        new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull = lock.newCondition();
  }
```
ArrayBlockingQueue is a blocking bounded FIFO queue

- It’s implemented using a dynamically sized array
- It has a ReentrantLock & two ConditionObjects

The ArrayBlockingQueue has a fixed-size capacity
public class ArrayBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
    java.io.Serializable {

    public ArrayBlockingQueue
        (int capacity, 
         boolean fair) {
        items = 
            new Object[capacity];
        lock = new ReentrantLock(fair);
        notEmpty = lock.newCondition();
        notFull = lock.newCondition();
    }

• ArrayBlockingQueue is a blocking bounded FIFO queue
  • It’s implemented using a dynamically sized array
• It has a ReentrantLock & two ConditionObjects

The “fair” parameter controls the order in which a group of threads can call methods on the queue

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/
ArrayBlockingQueue.html#ArrayBlockingQueue
public class ArrayBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, java.io.Serializable {
    
    public ArrayBlockingQueue (int capacity, boolean fair) {
        items = new Object[capacity];
        lock = new ReentrantLock(fair);
        notEmpty = lock.newCondition();
        notFull = lock.newCondition();
    }

    ...

If true then queue accesses for threads blocked on insertion or removal are processed in FIFO order, whereas if false access order is unspecified.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html#ArrayBlockingQueue
Applying Java ConditionObject in Practice

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - It’s implemented using a dynamically sized array
  - It has a ReentrantLock & two ConditionObjects

```java
public class ArrayBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
    java.io.Serializable {
    ...
    public ArrayBlockingQueue
        (int capacity,
         boolean fair) {
        items =
            new Object[capacity];
        lock = new ReentrantLock(fair);
        notEmpty = lock.newCondition();
        notFull = lock.newCondition();
    }
```

Both ConditionObjects share a common ReentrantLock returned via a factory method
Visualizing the Condition Object in Action
• ReentrantLock & Condition Objects implement the Monitor Object pattern

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

    final ReentrantLock lock;

    private final Condition notEmpty;
    private final Condition notFull;

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

    /** Main lock guarding access */
    final ReentrantLock lock;

    /** Condition for waiting takes */
    private final Condition notEmpty;

    /** Condition for waiting puts */
    private final Condition notFull;

    ...
Visualizing a Java Condition Object for Take ($T_1$)
Visualizing a Java ConditionObject for Take (T₁)

- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

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

// Called by thread T₁
String s = q.take();
```

*This call to the take() method blocks since the queue is initially empty*
Visualizing a Java Condition Object for Take ($T_1$)

- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

```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();
    }
}
```

When `take()` is called thread $T_1$ enters the monitor object if there's no contention of the monitor lock
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();
        }
    }

    ...
Visualizing a Java ConditionObject for Take ($T_1$)

- ReentrantLock & Condition Objects implement the Monitor Object pattern

```
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();
        }
    }
```

The Guarded Suspension pattern waits until the queue’s not empty

See en.wikipedia.org/wiki/Guarded_suspension
Visualizing a Java ConditionObject for Take ($T_1$)

- ReentrantLock & Condition Objects implement the Monitor Object pattern

```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();
    }
}
```

The call to await() atomically blocks $T_1$ & releases the lock
Visualizing a Java Condition Object for Put ($T_2$)
ReentrantLock & Condition
Objects implement the *Monitor Object* pattern.

ArrayBlockingQueue `q` =
`new ArrayBlockingQueue<> (10);`
...

// Called by thread T2
String `s` =
`new String("..."`);
...

`q.put(s);`
...

Thread T₂ puts a new string into the queue, which is currently empty &
which has thread T₁ waiting on the notEmpty ConditionObject.
ReentrantLock & Condition Objects implement the Monitor Object pattern

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

    public void put(E e) ... {
        final ReentrantLock lock =
            this.lock;
        lock.lockInterruptibly();
        try {
            while (count == items.length)
                notFull.await();
            insert(e);
            } finally {
                lock.unlock();
        }

    }
```

When put() is called thread \( T_2 \) enters the monitor object
• ReentrantLock & Condition Objects implement the *Monitor Object* pattern

---

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

    ...  
    public void put(E e) ... {
        ...  
        final ReentrantLock lock = this.lock;  
        lock.lockInterruptibly();  
        try {
            while (count == items.length)  
                notFull.await();  
            insert(e);  
        } finally {
            lock.unlock();  
        }
    }

Thread T₂ acquires the monitor lock & enters the critical section since there's no contention from other threads
Visualizing a Java ConditionObject for Put (T₂)

• ReentrantLock & Condition Objects implement the *Monitor Object* pattern

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

    ... public void put(E e) ... {
        ... final ReentrantLock lock =
            this.lock;
        lock.lockInterruptibly();
        try {
            while (count == items.length)
                notFull.await();
            insert(e);
        } finally {
            lock.unlock();
        }
    }

See [en.wikipedia.org/wiki/Guarded_suspension](en.wikipedia.org/wiki/Guarded_suspension)
```
Visualizing a Java ConditionObject for Put (T₂)

- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

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

    public void put(E e) ... {
        ... 
        final ReentrantLock lock =
            this.lock;
        lock.lockInterruptibly();
        try {
            while (count == items.length)
                notFull.await();
            insert(e);
        } finally {
            lock.unlock();
        }
    }

    // After the condition is satisfied the new element can be inserted into the queue
```
• ReentrantLock & Condition Objects implement the Monitor Object pattern

public class ArrayBlockingQueue<E> extends AbstractQueue<E>
    implements BlockingQueue<E>, java.io.Serializable {
    ...  
    private void insert(E x) {
        items[putIndex] = x;
        putIndex = inc(putIndex);
        ++count;
        notEmpty.signal();
    }

...
Visualizing a Java ConditionObject for Put (T₂)

- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

```java
public class ArrayBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, java.io.Serializable {
    ...
    private void insert(E x) {
        items[putIndex] = x;
        putIndex = inc(putIndex);
        ++count;
        notEmpty.signal();
    }
}
```

This method updates the state of the queue.
• ReentrantLock & Condition Objects implement the *Monitor Object* pattern

---

```
public class ArrayBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>, java.io.Serializable {
    ...
    private void insert(E x) {
        items[putIndex] = x;
        putIndex = inc(putIndex);
        ++count;
        notEmpty.signal();
    }
```

*It then signals the notEmpty condition object to indicate the queue’s no longer empty*
• ReentrantLock & Condition Objects implement the Monitor Object pattern

Visualizing a Java ConditionObject for Put (T2)

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

...  

public void put(E e) ... {

...  

final ReentrantLock lock =  
this.lock;  
lock.lockInterruptibly();  
try {
    while (count == items.length)  
        notFull.await();  
    insert(e);  
} finally {
    lock.unlock();  
}

The put() method then unlocks the monitor lock
Visualizing a Java Condition Object for Put ($T_2$)

- ReentrantLock & Condition Objects implement the Monitor Object pattern

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

    ... public void put(E e) ... {
    ...
        final ReentrantLock lock =
            this.lock;
        lock.lockInterruptibly();
        try {
            while (count == items.length) {
                notFull.await();
                insert(e);
            }
        } finally {
            lock.unlock();
        }
    }
```

*The put() method finally leaves the monitor*
Visualizing a Condition Object for Take \((T_1)\)
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();
    }
}

When insert() signals the notEmpty condition thread $T_1$ wakes up & returns in take()
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();  
    }
}

Before await() returns the monitor lock will be reacquired atomically
Visualizing a Java ConditionObject for Put ($T_1$)

- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

The *Guarded Suspension* pattern waits to see if the queue is no longer empty

```
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 [en.wikipedia.org/wiki/Guarded_suspension](en.wikipedia.org/wiki/Guarded_suspension)
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();
    }
}

When the condition is satisfied the extract() method is called
public class ArrayBlockingQueue<E> 
   extends AbstractQueue<E> 
   implements BlockingQueue<E>, 
   java.io.Serializable {

   ...

   private E extract() { 
      final Object[] items = 
         this.items;
      E x = 
         this.<E>cast 
            (items[takeIndex]);
      items[takeIndex] = null;
      takeIndex = inc(takeIndex);
      --count;
      notFull.signal();
      return x;
   }

   extract() assumes it’s called with the monitor lock held

• ReentrantLock & Condition Objects implement the Monitor Object pattern

Visualizing a Java ConditionObject for Put (T₁)
ReentrantLock & Condition Objects implement the Monitor Object pattern

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

    private E extract() {
        final Object[] items =
            this.items;
        E x =
            this.<E>cast
                (items[takeIndex]);
        items[takeIndex] = null;
        takeIndex = inc(takeIndex);
        --count;
        notFull.signal();
        return x;
    }

    extract() updates the state of the queue to remove the front item
```
public class ArrayBlockingQueue<E>  
   extends AbstractQueue<E>  
   implements BlockingQueue<E>, java.io.Serializable {

   ... 

   private E extract() {
      final Object[] items = 
          this.items; 
      E x =  
          this.<E>cast  
          (items[takeIndex]);  
      items[takeIndex] = null;  
      takeIndex = inc(takeIndex);  
      --count;  
      notFull.signal();  
      return x;  
   }

   It then signals the notFull CO to alert any thread waiting in put() that the queue’s not full

   • ReentrantLock & Condition Objects implement the Monitor Object pattern

   Visualizing a Java ConditionObject for Put (T₁)
Visualizing a Java ConditionObject for Put (T₁)

- ReentrantLock & Condition Objects implement the Monitor Object pattern

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

    ...

    private E extract() {
        final Object[] items =
            this.items;
        E x =
            this.<E>cast
            (items[takeIndex]);
        items[takeIndex] = null;
        takeIndex = inc(takeIndex);
        --count;
        notFull.signal();
        return x;
    }

    The item that's extracted is then returned to the caller of take()
```
Visualizing a Java ConditionObject for Put ($T_1$)

- ReentrantLock & Condition Objects implement the Monitor Object pattern

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

The `take()` method then unlocks the monitor lock.
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();
    }
  }

The take() method then finally leaves the monitor
Visualizing a Java ConditionObject for Put ($T_1$)

- ReentrantLock & Condition Objects implement the Monitor Object pattern

```
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();
        }
    }

    ... 
```

This example is complex due to the concurrent coordination between threads & the “moving parts” between the lock & condition objects!
End of Java ConditionObject: Example Application