Java ConditionObject (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 what condition variables are & what pattern they implement
- Recognize how condition variables are often applied in practice
- Be aware of a human known use of condition variables
- Learn how Java ConditionObject enables concurrent programs to have multiple wait-sets per user-defined object
Overview of Java ConditionObject
Overview of Java ConditionObject

- ConditionObject provides the condition variable abstraction

```java
public class ConditionObject
    implements Condition, java.io.Serializable {
    ...
```

**Class AbstractQueuedSynchronizer.ConditionObject**

```java
java.lang.Object
    java.util.concurrent.locks.AbstractQueuedSynchronizer.ConditionObject

All Implemented Interfaces:
    Serializable, Condition

Enclosing class:
    AbstractQueuedSynchronizer

public class AbstractQueuedSynchronizer.ConditionObject
    extends Object
    implements Condition, Serializable

Condition implementation for a AbstractQueuedSynchronizer serving as the basis of a Lock implementation.

Method documentation for this class describes mechanics, not behavioral specifications from the point of view of Lock and Condition users. Exported versions of this class will in general need to be accompanied by documentation describing condition semantics that rely on those of the associated AbstractQueuedSynchronizer.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html
Overview of Java ConditionObject

- ConditionObject provides the condition variable abstraction
- Implements Condition interface

```
public class ConditionObject
    implements Condition, java.io.Serializable {
...
```

### Interface Condition

**All Known Implementing Classes:**
- AbstractQueuedLongSynchronizer.ConditionObject,
  AbstractQueuedSynchronizer.ConditionObject

#### public interface Condition

Condition factors out the Object monitor methods (wait, notify and notifyAll) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations. Where a Lock replaces the use of synchronized methods and statements, a Condition replaces the use of the Object monitor methods.

Conditions (also known as condition queues or condition variables) provide a means for one thread to suspend execution (to "wait") until notified by another thread that some state condition may now be true. Because access to this shared state information occurs in different threads, it must be protected, so a lock of some form is associated with the condition. The key property that waiting for a condition provides is that it **atomically releases the associated lock and suspends the current thread, just like Object.wait.**

A Condition instance is intrinsically bound to a lock. To obtain a Condition instance for a particular Lock instance use its `newCondition()` method.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Condition.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Condition.html)
Overview of Java ConditionObject

- ConditionObject is nested within the AbstractQueuedSynchronizer class.
- This framework is used by Java synchronizers that rely on FIFO wait queues.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html
Overview of Java ConditionObject

- A ConditionObject provides a “wait queue” of nodes

See gee.cs.oswego.edu/dl/papers/aqs.pdf
Overview of Java ConditionObject

- A ConditionObject provides a “wait queue” of nodes
- Enables a set of threads (i.e., the “wait set”) to coordinate their interactions
Overview of Java ConditionObject

- A ConditionObject provides a “wait queue” of nodes
- Enables a set of threads (i.e., the “wait set”) to coordinate their interactions
  - e.g., by selecting the order & conditions under which they run

```java
<<Java Class>>
ConditionObject
- firstWaiter: Node
- lastWaiter: Node
- ConditionObject()
- await():void
- awaitUninterruptibly():void
- await(long, TimeUnit):boolean
- signal():void
  - doSignal(Node):void
  - signalAll():void
  - doSignalAll(Node):void

<<Java Class>>
Node
- EXCLUSIVE: Node
- SHARED: Node
- prev: Node
- next: Node
- thread: Thread
- nextWaiter: Node
- Node()
```

```java
<<Java Class>>
AbstractQueuedSynchronizer
- state: int
- head: Node
- tail: Node
- getState():int
- setState(int):void
- AbstractQueuedSynchronizer()
- compareAndsetState(int,int):boolean
- tryAcquire(int):boolean
- tryRelease(int):boolean
- tryAcquireShared(int):int
- tryReleaseShared(int):boolean
- isHeldExclusively():boolean
- acquire(int):void
- acquireInterruptibly(int):void
- tryAcquireNanos(int,long):boolean
- release(int):boolean
- acquireShared(int):void
- acquireSharedInterruptibly(int):void
- tryAcquireSharedNanos(int,long):boolean
- releaseShared(int):boolean
```
Overview of Java ConditionObject

- A ConditionObject is *always* used with a lock

```
Overview of Java
ConditionObject

Consumer

ArrayBlocking Queue
  put()
  take()

Producer

ConditionObject
  await()
  signal()
  signalAll()

Reentrant Lock
  lock()
  unlock()
  newCondition()
```

See earlier part on "Java ReentrantLock"
Overview of Java ConditionObject

- A ConditionObject is *always* used with a lock
- This lock protects shared state in a condition expression from concurrent manipulation

```
Consumer

ArrayBlockingQueue
  put()
  take()

Producer

ConditionObject
  await()
  signal()
  signalAll()

ReentrantLock
  lock()
  unlock()
  newCondition()
```

<<uses>>
<<uses>>
A ConditionObject is *always* used with a lock

- This lock protects shared state in a condition expression from concurrent manipulation

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html#newCondition](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html#newCondition)

*newCondition() is a factory method that returns a ConditionObject that can be used with this lock*
Overview of Java ConditionObject

- Both ReentrantLock & ConditionObject have internal queues

```
Consumer

ArrayBlockingQueue
  put()
  take()
  take()

Producer

ConditionObject
  await()
  signal()
  signalAll()

ReentrantLock
  lock()
  unlock()
  newCondition()
```

<<uses>>

<<uses>>
Overview of Java ConditionObject

- Both ReentrantLock & ConditionObject have internal queues

Queues up threads that are waiting to acquire the lock
Overview of Java ConditionObject

• Both ReentrantLock & ConditionObject have internal queues

Queues up threads waiting for some condition(s) to become true
Overview of Java ConditionObject

- User-defined Java objects can have multiple ConditionObjects (COs)

Two COs: mNotEmpty & mNotFull
Overview of Java ConditionObject

- User-defined Java objects can have multiple ConditionObjects (COs)
- Multiple COs enable more sophisticated & efficient ways to coordinate multiple threads

```java
Consumer
    take()

ArrayBlockingQueue
    put()
    take()

Producer
    put()

ConditionObject
    await()
    signal()
    signalAll()

ReentrantLock
    lock()
    unlock()
    newCondition()
```

<<uses>>
Overview of Java ConditionObject

- User-defined Java objects can have multiple ConditionObjects (COs)
- Multiple COs enable more sophisticated & efficient ways to coordinate multiple threads
- e.g., multiple wait-sets per user object that share a lock & are notified on different conditions

See stackoverflow.com/questions/18490636/condition-give-the-effect-of-having-multiple-wait-sets-per-object
Overview of Java ConditionObject

• In contrast, Java’s built-in monitor objects only support one monitor condition.

i.e., there’s just a single “wait queue”

See upcoming lesson on “Java Built-in Monitor Objects”
Overview of Java ConditionObject

- In contrast, Java’s built-in monitor objects only support one monitor condition.
- Yields inefficient programs that require excessive notifications & use of notifyAll().

See [www.dre.vanderbilt.edu/~schmidt/C++2Java.html#concurrency](http://www.dre.vanderbilt.edu/~schmidt/C++2Java.html#concurrency)
• In contrast, Java’s built-in monitor objects only support one monitor condition

• Yields inefficient programs that require excessive notifications & use of notifyAll()

• e.g., producers & consumers must both wake up on every change to the queue, even if a given thread can’t proceed

```java
synchronized(this) {
    while (mList.isEmpty())
        wait();
    notifyAll();
    return mList.poll();
}
```

Key Methods of Java
ConditionObject
Key Methods of Java ConditionObject

- Its key methods allow threads to wait & notify each other

```java
public class ConditionObject implements Condition, java.io.Serializable {

    ... 

    /** Implement interruptible condition wait. */
    public final void await()
        throws InterruptedException
    { ... }

    /** Wakeup the longest waiting thread. */
    public final void signal()
    { ... }

    /** Wakeup all waiting threads. */
    public final void signalAll()
    { ... }

    ...
}
```
Its key methods allow threads to wait & notify each other

```java
public class ConditionObject
    implements Condition,
    java.io.Serializable {

    ... 

    /** Implement interruptible condition wait. */
    public final void await()
        throws InterruptedException
    { ... }

    /** Wakeup the longest waiting thread. */
    public final void signal()
    { ... }

    /** Wakeup all waiting threads. */
    public final void signalAll()
    { ... }

    ...
}
```

Method names are similar to Java's built-in monitor object methods, but Java Object final methods can't be overridden

See lesson on “Java Built-in Monitor Objects”
Key Methods of Java ConditionObject

- Its key methods allow threads to wait & notify each other

```java
class ConditionObject implements Condition, java.io.Serializable {

  void await() throws InterruptedException {
  }

  void signal() {
  }

  void signalAll() {
  }
}
```

Methods are implemented via the AbstractQueuedSynchronizer framework

See [gee.cs.oswego.edu/dl/papers/aqs.pdf](http://gee.cs.oswego.edu/dl/papers/aqs.pdf)
Key Methods of Java ConditionObject

- Its key methods allow threads to wait & notify each other
- `await()` suspends the calling thread until it’s signaled

```java
public class ConditionObject
    implements Condition, java.io.Serializable {
    ...
    /** Implement interruptible condition wait. */
    public final void await() ...
    { ... }
    ...
}

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html#await
Its key methods allow threads to wait & notify each other.

- `await()` suspends the calling thread until it’s signaled.
- The thread is “parked” on the condition object’s queue.

```java
public class ConditionObject implements Condition, java.io.Serializable {

    /** Implement interruptible condition wait. */
    public final void await() {
        ...
    }

    See [www.docjar.com/docs/api/sun/misc/Unsafe.html#park(boolean, long)](http://www.docjar.com/docs/api/sun/misc/Unsafe.html#park(boolean, long))
```
Key Methods of Java ConditionObject

- Its key methods allow threads to wait & notify each other
  - `await()` suspends the calling thread until it’s signaled
  - `signal()` moves the longest waiting thread from the queue for this condition object to the queue for the owning lock

```java
class ConditionObject implements Condition, java.io.Serializable {
    ... 
    /** Wakeup longest waiting thread. */
    public final void signal() {
        ... 
    }
    ...
}
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html#signal](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html#signal)
Key Methods of Java ConditionObject

- Its key methods allow threads to wait & notify each other
  - `await()` suspends the calling thread until it’s signaled
  - `signal()` moves the longest waiting thread from the queue for this condition object to the queue for the owning lock
  - `signalAll()` moves all threads from the `ConditionObject`’s queue to owning lock’s queue

```java
public class ConditionObject implements Condition, java.io.Serializable {
    public final void signalAll() {
        /*
        ** Wakeup all waiting threads. */
        public final void signalAll()
        { ... }
    ...
}
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html#signalAll
Key Methods of Java ConditionObject

- Its key methods allow threads to wait & notify each other
  - `await()` suspends the calling thread until it’s signaled
  - `signal()` moves the longest waiting thread from the queue for this condition object to the queue for the owning lock
  - `signalAll()` moves all threads from the ConditionObject’s queue to owning lock’s queue
  - `signalAll()` may cause the “thundering herd” problem, so use it sparingly!!

```
public class ConditionObject extends Object implements Condition, java.io.Serializable {

    public final void signalAll() {
        // Implementation...
    }

    /** Wakeup all waiting threads. */
    public final void signalAll() {
        // Implementation...
    }

See en.wikipedia.org/wiki/Thundering_herd_problem
Other Methods of Java
ConditionObject
**Other Methods of Java `ConditionObject`**

- `ConditionObject` has several `await()` methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void await()</code></td>
<td>Causes the current thread to wait until it is signalled or interrupted</td>
</tr>
<tr>
<td><code>boolean await(long time, TimeUnit unit)</code></td>
<td>Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses</td>
</tr>
<tr>
<td><code>long awaitNanos(long nanosTimeout)</code></td>
<td>Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses</td>
</tr>
<tr>
<td><code>void awaitUninterruptibly()</code></td>
<td>Causes the current thread to wait until it is signalled</td>
</tr>
<tr>
<td><code>boolean awaitUntil(Date deadline)</code></td>
<td>Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses</td>
</tr>
</tbody>
</table>
ConditionObject has several `await()` methods

- e.g., interruptible, non-interruptible, & timed operations

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td><code>await()</code></td>
<td>Causes the current thread to wait until it is signalled or interrupted</td>
</tr>
<tr>
<td>boolean</td>
<td><code>await(long time, TimeUnit unit)</code></td>
<td>Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses</td>
</tr>
<tr>
<td>long</td>
<td><code>awaitNanos(long nanosTimeout)</code></td>
<td>Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses</td>
</tr>
<tr>
<td>void</td>
<td><code>awaitUninterruptibly()</code></td>
<td>Causes the current thread to wait until it is signalled</td>
</tr>
<tr>
<td>boolean</td>
<td><code>awaitUntil(Date deadline)</code></td>
<td>Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses</td>
</tr>
</tbody>
</table>
Other Methods of Java ConditionObject

- ConditionObject has several `await()` methods
  - e.g., interruptible, non-interruptible, & timed operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td><code>await()</code> – Causes the current thread to wait until it is signalled or interrupted</td>
</tr>
<tr>
<td>boolean</td>
<td><code>await(long time, TimeUnit unit)</code> – Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses</td>
</tr>
<tr>
<td>long</td>
<td><code>awaitNanos(long nanosTimeout)</code> – Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses</td>
</tr>
<tr>
<td>void</td>
<td><code>awaitUninterruptibly()</code> – Causes the current thread to wait until it is signalled</td>
</tr>
<tr>
<td>boolean</td>
<td><code>awaitUntil(Date deadline)</code> – Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses</td>
</tr>
</tbody>
</table>

Unlike Java's built-in monitor object timed `wait()` calls, these timed `await*()` calls give a sensible return value.

See stackoverflow.com/questions/3397722/how-to-differentiate-when-waitlong-timeout-exit-for-notify-or-timeout
End of Java Condition Object (Part 3)
Learning Objectives in this Part of the Lesson

- Understand what condition variables are & what pattern they implement
- Recognize how condition variables are often applied in practice
- Be aware of a human known use of condition variables
- Learn how Java ConditionObject enables concurrent programs to have multiple wait-sets per user-defined object
- Know how to use Java ConditionObjects in practice
Using 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**

```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)
• ArrayBlockingQueue is a blocking bounded 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](docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html)
Using 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 {
```

---

**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
Using 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;

    ...
```
Using Java ConditionObject in Practice

- ArrayBlockingQueue is a blocking bounded FIFO queue
- It's implemented using a dynamically sized array

Object state that needs to be protected from race conditions & 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;

    ...
```

9
Using 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 {

    private final ReentrantLock lock;
    private final Condition notEmpty;
    private final Condition notFull;

    // Constructor...

    // Getters and Setters...

    // Methods...

    // Example of use...

    int putItem(E item) {
        synchronized (lock) {
            lock.lock();
            try {
                while (isFull()) {
                    notFull.await();
                }
                // Put item...
                return 0;
            } finally {
                lock.unlock();
            }
        }
    }

    int takeItem() {
        synchronized (lock) {
            lock.lock();
            try {
                while (isEmpty()) {
                    notEmpty.await();
                }
                // Take item...
                return 0;
            } finally {
                lock.unlock();
            }
        }
    }

    // Other methods...

    // Example usage...

    // End of example...

    // End of class...
```

*Used to protect the object state from race conditions*
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;

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

    final ReentrantLock lock;
    private final ConditionnotEmpty;
    private final ConditionnotFull;

    /** 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 wakups & checking

See stackoverflow.com/questions/18490636/condition-give-the-effect-of-having-multiple-wait-sets-per-object
public class ArrayBlockingQueue<br>   extends AbstractQueue<br>   implements BlockingQueue,<br>   java.io.Serializable {

    public ArrayBlockingQueue<br>        (int capacity,<br>            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 an dynamically sized array
• It has a ReentrantLock & two ConditionObjects
public class ArrayBlockingQueue&lt;E&gt; extends AbstractQueue&lt;E&gt; implements BlockingQueue&lt;E&gt;, java.io.Serializable {

... public ArrayBlockingQueue (int capacity,

boolean fair) {

items =

new Object[capacity];
lock = new ReentrantLock(fair);
notEmpty = lock.newCondition();
notFull = lock.newCondition();

}
public class ArrayBlockingQueue
    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();
    }

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
Using 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();
    }

    // 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
Using 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
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 ConditionnotEmpty;

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

    ...
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;

    The steps visualized next apply to both the Monitor Object pattern & Java condition objects

    ...
Visualizing a Java Condition Object for Take \((T_1)\)
ReentrantLock & Condition Objects implement the Monitor Object pattern

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

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

This call to the take() method blocks since the queue is initially empty
ReentrantLock & Condition Objects implement the Monitor Object pattern.

Visualizing a Java Condition Object for Take ($T_1$)

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

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

Thread $T_1$ then acquires the lock & enters the critical section since there’s no contention from other threads
```
Visualizing a Java ConditionObject for Take ($T_1$)

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

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

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

See en.wikipedia.org/wiki/Guarded_suspension
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 call to await() atomically blocks \( T_1 \) & releases the lock.
Visualizing a Java Condition Object for Put ($T_2$)
Visualizing a Java ConditionObject for Put ($T_2$)

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

ArrayBlocking Queue

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

Thread $T_2$ puts a new string into the queue, which is currently empty & which has thread $T_1$ waiting on the `notEmpty` ConditionObject.
ReentrantLock & Condition Objects implement the Monitor Object pattern.

When put() is called thread $T_2$ enters the monitor object.
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();
        }
    }
```

Thread T₂ acquires the monitor lock & enters the critical section since there’s no contention from other threads
ReentrantLock & Condition Objects implement the Monitor Object pattern.

The Guarded Suspension pattern waits until queue's not full.

```
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
Visualizing a Java Condition Object 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.
```
Visualizing a Java ConditionObject for Put (T₂)

- 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();
    }

    The insert() method is not synchronized since it must be called with the lock held

33
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*
Visualizing a Java ConditionObject for Put (T₂)

- 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

```
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 ConditionObject for Put (T2)

- 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 ($T_1$)
• ReentrantLock & Condition Objects implement the Monitor Object pattern

Visualizing a Java ConditionObject for Put ($T_1$)

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

    Before await() returns
    the monitor lock will be
    reacquired atomically
```

ArrayBlocking Queue

- `notFull`
- `notEmpty`
- `Critical Section`
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
• ReentrantLock & Condition Objects implement the Monitor Object pattern

Visualizing a Java ConditionObject for Put (T₁)

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
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;
    }

    extract() assumes it's called with the monitor lock held
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 condition object to alert any thread waiting in put() that the queue's not full
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 take() caller
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();
        }
    }

    // The take() method then unlocks the monitor lock
```

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 {

    ...

    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.
```
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 Condition Object (Part 4)
Learning Objectives in this Part of the Lesson

- Understand what condition variables are & what pattern they implement
- Recognize how condition variables are often applied in practice
- Be aware of a human known use of condition variables
- Learn how Java ConditionObject enables concurrent programs to have multiple wait-sets per user-defined object
- Know how to use Java ConditionObjects in practice
- Appreciate ConditionObject usage considerations
ConditionObject
Usage Considerations
ConditionObject Usage Considerations

- ConditionObject is a highly flexible synchronization mechanism.
• ConditionObject is a highly flexible synchronization mechanism
• Allows threads to suspend & resume their execution based on shared state

**Usage Considerations**

- **e.g., threads T₁ & T₂ can take turns sharing a critical section**

**Diagram:**
- Cond Obj₁
- Cond Obj₂
- Critical Section
- Lock
- Thread T₁ accesses the critical section, while thread T₂ waits
- T₁
- T₂
- T₃
ConditionObject is a highly flexible synchronization mechanism

- Allows threads to suspend & resume their execution based on shared state

**Usage Considerations**

e.g., threads $T_1$ & $T_2$ can take turns sharing a critical section

Thread $T_2$ accesses the critical section, while thread $T_1$ waits
ConditionObject Usage Considerations

- ConditionObject is a highly flexible synchronization mechanism
  - Allows threads to suspend & resume their execution based on shared state
- A user object can define multiple ConditionObjects
ConditionObject Usage Considerations

- ConditionObject is a highly flexible synchronization mechanism
  - Allows threads to suspend & resume their execution based on shared state
- A user object can define multiple ConditionObjects
- Each ConditionObject can provide a separate “wait set”
ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
• However, a ConditionObject must be used carefully to avoid problems
• It should (almost) always be waited upon in a loop

```java
public class ArrayBlockingQueue<E> {
    ... {
        ...
        public E take() ... {
            final ReentrantLock lock = this.lock;
            lock.lockInterruptibly();
            try {
                while (count == 0)
                    notEmpty.await();
                return extract();
            } finally {
                lock.unlock();
            }
        }
    }
}```
ConditionObject Usage Considerations

• However, a ConditionObject must be used carefully to avoid problems
  • It should (almost) always be waited upon in a loop
  • (Re)test state that’s being waited for since it may change due to non-determinism of concurrency

See docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

```java
public class ArrayBlockingQueue<E> {
    ...
    ...
    public E take() {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == 0)
                notEmpty.await();
            return extract();
        } finally {
            lock.unlock();
        }
    }
}
```
ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
- It should (almost) always be waited upon in a loop
- (Re)test state that’s being waited for since it may change due to non-determinism of concurrency
- Guard against spurious wakeups

```java
public class ArrayBlockingQueue<E> {
    ... {
        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/Spurious_wakeup](en.wikipedia.org/wiki/Spurious_wakeup)
ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
- It should (almost) always be waited upon in a loop
  - (Re)test state that’s being waited for since it may change due to non-determinism of concurrency
- Guard against spurious wakeups
  - A thread might be awoken from its waiting state even though no thread signaled the CO

```java
public class ArrayBlockingQueue<E> {
    ... {
        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/Spurious_wakeup](en.wikipedia.org/wiki/Spurious_wakeup)
ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
  - It should (almost) always be waited upon in a loop
  - It is always used in conjunction with a lock
ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
  - It should (almost) always be waited upon in a loop
  - It is always used in conjunction with a lock
  - Needed to avoid the “lost wakeup problem”

- A thread calls `signal()` or `signalAll()`
- Another thread is between the test of the condition & the call to `await()`
- No threads are waiting

See docs.oracle.com/cd/E19253-01/816-5137/sync-30
ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
  - It should (almost) always be waited upon in a loop
  - It is always used in conjunction with a lock
    - Needed to avoid the "lost wakeup problem"
  - `await()` internally releases & reacquires its associated lock!
**ConditionObject Usage Considerations**

- However, a ConditionObject must be used carefully to avoid problems
  - It should (almost) always be waited upon in a loop
  - It is always used in conjunction with a lock
  - Choosing between `signal()` & `signalAll()` can be subtle
• However, a ConditionObject must be used carefully to avoid problems
- It should (almost) always be waited upon in a loop
- It is always used in conjunction with a lock
- Choosing between signal() & signalAll() can be subtle
  - Using signal() is more efficient & avoids the “Thundering Herd” problem.

See en.wikipedia.org/wiki/Thundering_Herd_problem
ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
  - It should (almost) always be waited upon in a loop
  - It is always used in conjunction with a lock
  - Choosing between signal() & signalAll() can be subtle

**Uniform waiters**
- Only one condition expression that `await()` is waiting for is associated with the ConditionObject wait set & each thread executes the same logic when returning from `await()`

**One-in & one-out**
- A signal() on the ConditionObject enables at most one thread to proceed

**Conditions under which signal() can be used**

The implementation of Java ArrayBlockingQueue demonstrates the issue

See Part 4 of "Java ConditionObject"
However, a ConditionObject must be used carefully to avoid problems:

- It should (almost) always be waited upon in a loop.
- It is always used in conjunction with a lock.
- Choosing between signal() & signalAll() can be subtle.

**ConditionObject Usage Considerations**

**Uniform waiters**
- Only one condition expression that `await()` is waiting for is associated with the ConditionObject wait set & each thread executes the same logic when returning from `await()`.

**One-in & one-out**
- A `signal()` on the ConditionObject enables at most one thread to proceed.

**Conditions under which signal() can be used**

```java
public E take() ... {
    ...
    while (count == 0) {
        notEmpty.await();
        return extract();
    }
    ...
}
```

See Part 4 of “Java ConditionObject”
However, a ConditionObject must be used carefully to avoid problems:

- It should (almost) always be waited upon in a loop.
- It is always used in conjunction with a lock.
- Choosing between signal() & signalAll() can be subtle.

**Conditions under which signal() can be used**

```java
private void insert(E x) {
    items[putIndex] = x;
    putIndex = inc(putIndex);
    ++count;
    notEmpty.signal();
}
```
ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
  - It should (almost) always be waited upon in a loop
  - It is always used in conjunction with a lock
  - Choosing between `signal()` & `signalAll()` can be subtle

<table>
<thead>
<tr>
<th></th>
<th>Uniform waiters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Only one condition expression that <code>await()</code> is waiting for is associated with</td>
</tr>
<tr>
<td></td>
<td>the ConditionObject wait set &amp; each thread executes the same logic when</td>
</tr>
<tr>
<td></td>
<td>returning from <code>wait()</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>One-in &amp; one-out</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A signal() on the ConditionObject enables at most one thread to proceed</td>
</tr>
</tbody>
</table>

Conditions under which `signal()` can be used

The implementation of Java ArrayBlockingQueue satisfies both conditions
ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
  - It should (almost) always be waited upon in a loop
  - It is always used in conjunction with a lock
  - Choosing between signal() & signalAll() can be subtle
  - ConditionObject inherits the wait(), notify(), & notifyAll() methods from Java Object!!

Do *not* mix & match these methods!!
ConditionObject Usage Considerations

- ConditionObject is used in java.util.concurrent & java.util.concurrent.locks

package
java.util.concurrent.locks

Added in API level 1

Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors. The framework permits much greater flexibility in the use of locks and conditions, at the expense of more awkward syntax. The Lock interface supports locking disciplines that differ in semantics (reentrant, fair, etc), and that can be used in non-block-structured contexts including hand-over-hand and lock reordering algorithms. The main implementation is ReentrantLock.

package
java.util.concurrent

Added in API level 1

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the java.util.concurrent.locks and java.util.concurrent.atomic packages.
ConditionObject Usage Considerations

- ConditionObject is used in java.util.concurrent & java.util.concurrent.locks
- However, it’s typically hidden within higher-level abstractions
**ConditionObject Usage Considerations**

- ConditionObject is used in `java.util.concurrent` & `java.util.concurrent.locks`
- However, it’s typically hidden within higher-level abstractions
  - e.g., `ArrayBlockingQueue` & `LinkedBlockingQueue`

---

**Diagram:**

- **Consumer**
  - `take()`

- **Producer**
  - `put()`

- **ArrayBlockingQueue**
  - `put()`
  - `take()`
  - Uses 2

- **ConditionVariable**
  - `await()`
  - `signal()`
  - `signalAll()`
  - Uses

- **Lock**
  - `lock()`
  - `unlock()`
  - Uses

---

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)
End of Java Condition Object (Part 5)