Java ConditionObject (Part 4)

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
- Know how to use Java ConditionObjects in practice
Using ConditionObject in Practice
Using ConditionObject in Practice

- ArrayBlockingQueue is a blocking bounded FIFO queue

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

**Class ArrayBlockingQueue<E>**

```
java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractQueue<E>
      java.util.concurrent.ArrayBlockingQueue<E>
```

Type Parameters:

- **E** - the type of elements held in this collection

All Implemented Interfaces:

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

```
public class ArrayBlockingQueue<E>
extends AbstractQueue<E>
implements BlockingQueue<E>, Serializable
```

A bounded blocking queue backed by an array. This queue orders elements FIFO (first-in-first-out). The head of the queue is that element that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html)
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](http://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html)
Using ConditionObject in Practice

- 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](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)
Using 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
Using ConditionObject in Practice

- 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 {

    final Object[] items;

    int takeIndex;

    int putIndex;

    ...}
```
• 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;
...
```

Object state that needs to be protected from race conditions & coordinate concurrent put() & take() calls
ArrayBlockingQueue is a blocking bounded FIFO queue
• It’s implemented using a dynamically sized array
• It has a ReentrantLock & two ConditionObjects

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

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

    final ReentrantLock lock;

    private final Condition notEmpty;
    private 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;

    ...
```
Using ConditionObject in Practice

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

See earlier lesson on “Java ReentrantLock” for the initial coverage of ArrayBlockingQueue

```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;

    ...

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

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

    /** Condition for waiting puts */
    private final Condition notFull;
    ...
```
Using 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
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;
    ...
}
```

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();
    }
```
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();
    }

The ArrayBlockingQueue has a fixed-size capacity
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
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();
    }
    ...
Visualizing the ConditionObject in Action
Visualizing the ConditionObject in Action

- ReentrantLock & Condition Objects implement the Monitor Object pattern

```
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;

    ...  

    See [www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf](www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf)
```
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 the ConditionObject 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 take() will block since the queue is initially empty
Visualizing the 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();
    }
  }

When take() is called thread $T_1$ enters the monitor object
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 T1 then acquires the lock & enters the critical section since there’s no contention from other threads.
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 is applied here to wait until the queue is no longer empty (i.e., count is greater than 0)
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 the ConditionObject for Put \( (T_2) \)
• ReentrantLock & Condition Objects implement the *Monitor Object* pattern

```
// 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.

Visualizing the ConditionObject for Put (T₂)

When put() is called thread T₂ enters the monitor object.
• ReentrantLock & Condition

Objects implement the

Monitor Object pattern

---

**Visualizing the ConditionObject for Put (T₂)**

**ArrayBlockingQueue**

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.

Visualizing the ConditionObject for Put (T₂)

- Critical Section
- notFull
- notEmpty
- Running Thread

ArrayBlockingQueue

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 Guarded Suspension pattern is applied here to wait until the queue is not full (i.e., count is less than length)
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();
        }
    }

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();
    }
```

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

This method updates the state of the queue.
Visualizing the ConditionObject for Put ($T_2$)

- 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();
    }
}
```

It then signals the notEmpty ConditionObject to indicate that the queue is no longer empty.
- 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(); }
    }

The put() method then unlocks the monitor lock
```
• 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(); }  
    }  

    ...  

}
Visualizing the ConditionObject for Take ($T_1$)
Visualizing the 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();
        }
    }
}
```

When put()/insert() signal the notEmpty condition that causes thread T₁ to wakeup & return in the take() method
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();
    }
}
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 will again be employed to see if the queue is no longer empty (i.e., count greater than 0)
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 the ConditionObject for Put (T₁)

  When the condition is satisfied the extract() method is called

  • 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;
    }

    // Diagram:
    // ArrayBlockingQueue
    // Critical Section
    // notFull
    // lock
    // T1 Running Thread
    // notEmpty

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

    The extract() method assumes its
    is 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;
}

This method 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;
    }

• ReentrantLock & Condition

Objects implement the
Monitor Object pattern

Visualizing the ConditionObject for Put (T₁)

It then signals the notFull ConditionObject to alert any thread waiting in put() that the queue is no longer full
Visualizing the 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 {

    ... 

    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 extracted item is returned to the take() caller
Visualizing the 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 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 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 & ConditionObjects!
End of Java ConditionObject (Part 4)
Java ConditionObject (Part 5)

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
- 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 Usage Considerations

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

Example usage:
- Threads $T_1$ & $T_2$ can take turns sharing a critical section.
ConditionObject Usage Considerations

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

*Example:* threads $T_1$ & $T_2$ can take turns sharing a critical section
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
ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
- It should 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();
            }
        }
    }
}
```
• However, a ConditionObject must be used carefully to avoid problems
• It should 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
ConditionObject Usage Considerations

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

See docs.oracle.com/cd/E19455-01/806-5257/sync-30/index.html
ConditionObject Usage Considerations

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

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

Use signal() if possible since it is more efficient & avoids the “Thundering Herd” problem.
**ConditionObject Usage Considerations**

- However, a ConditionObject must be used carefully to avoid problems
  - It should 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>Uniform waiters</th>
<th>Only one condition expression that <code>await()</code> is waiting for is associated with the ConditionObject wait set &amp; each thread executes the same logic when returning from <code>await()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>One-in &amp; one-out</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*
### ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
  - It should 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>Uniform waiters</th>
<th>Only one condition expression that <code>await()</code> is waiting for is associated with the ConditionObject wait set &amp; each thread executes the same logic when returning from <code>await()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>One-in &amp; one-out</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*
ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
  - It should 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>Uniform waiters</th>
<th>Only one condition expression that await() is waiting for is associated with the ConditionObject wait set &amp; each thread executes the same logic when returning from wait()</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-in &amp; one-out</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 Java ArrayBlockingQueue class implementation satisfies both conditions
ConditionObject Usage Considerations

- However, a ConditionObject must be used carefully to avoid problems
  - It should 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

```java
package java.util.concurrent.locks

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.
```

```java
package java.util.concurrent

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
  • e.g., ArrayBlockingQueue & LinkedBlockingQueue

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html
End of Java ConditionObject (Part 5)