Java ConditionObject

Usage Considerations

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 ConditionObjects in practice
- Appreciate ConditionObject usage considerations
Java ConditionObject
Usage Considerations
• ConditionObject is a highly flexible synchronization mechanism
Java ConditionObject Usage Considerations

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

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

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

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

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

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

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

- However, a ConditionObject must be used carefully to avoid problems
Java ConditionObject Usage Considerations

- 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();
        }
    }
}
```
Java 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.

```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 [docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html](docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html)
Java 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();
    }
}
```

A thread might be awoken from its waiting state even though no thread signaled the CO

See [en.wikipedia.org/wiki/Spurious_wakeup](en.wikipedia.org/wiki/Spurious_wakeup)
Java 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
Java 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](https://docs.oracle.com/cd/E19253-01/816-5137-sync-30)
Java 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!
Java 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
Java 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
    • Using signal() is more efficient & avoids the “Thundering Herd” problem..

See en.wikipedia.org/wiki/Thundering_herd_problem
Java 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 this issue**

See earlier discussion in "Java ConditionObject: Example Application"
Java 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

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

See earlier discussion in "Java ConditionObject: Example Application"
Java 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>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 await()</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

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

See earlier discussion in “Java ConditionObject: Example Application’
Java 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

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

<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>

Java ArrayBlockingQueue satisfies both conditions
Java 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!!!
Java ConditionObject Usage Considerations

- Name condition object fields to reflect their usage

- Used to wait until the condition is not empty
- Used to wait until the condition is not full
Java 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.
```
Java ConditionObject Usage Considerations

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

```mermaid
diagram
    title Java ConditionObject
    Container "Consumer" -> Container "ArrayBlocking Queue"
    Container "Producer" -> Container "ArrayBlocking Queue"
    Container "ConditionObject" -> Container "ReentrantLock"
    Container "ArrayBlocking Queue" uses 2 Container "ConditionObject"
    Container "ArrayBlocking Queue" uses Container "ReentrantLock"
    Container "ConditionObject" uses Container "ReentrantLock"
```
End of Java ConditionObject
Usage Considerations