Java ConditionObject (Part 1)
Learning Objectives in this Part of the Lesson

- Understand what condition variables are & what pattern they implement

```
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```
Learning Objectives in this Part of the Lesson

- Understand what condition variables are & what pattern they implement

```java
Lock l = new Lock();
Condition cond = l.newCondition();
...
cond.lock();
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

Condition variables are tricky, so you might want to rewatch this lesson & read the links carefully.
Overview of Condition Variables
Overview of Condition Variables

- A CV is a synchronizer that allows a thread to (repeatedly) suspend its execution until a condition becomes true

See blog.dcoles.net/2012/02/understanding-how-to-use-condition.html

Wheel of Pain – Conan the Barbarian
Overview of Condition Variables

- A CV is a synchronizer that allows a thread to (repeatedly) suspend its execution until a condition becomes true.
- A thread whose execution is suspended on a CV is said to be “blocked” on the CV.

Tree of Woe – Conan the Barbarian
Overview of Condition Variables

- A CV is implemented as a queue of threads that wait for a certain condition to become true.

```java
Lock l = new Lock();
Condition cond = l.newCondition();
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

See [en.wikipedia.org/wiki/Monitor_(synchronization)#Condition_variables](en.wikipedia.org/wiki/Monitor_(synchronization)#Condition_variables)
Overview of Condition Variables

- A CV is implemented as a queue of threads that wait for a certain condition to become true.
- Often used when *mutual exclusion* is insufficient to ensure *coordination*.
Overview of Condition Variables

• A CV is implemented as a queue of threads that wait for a certain condition to become true

• Often used when *mutual exclusion* is insufficient to ensure *coordination*

• e.g., what to do when a thread encounters shared state that it can't do any work on (yet)
Implementing Guarded Suspension with CVs
Implementing Guarded Suspension with CVs

- CVs are most often used to implement the *Guarded Suspension* pattern

See [en.wikipedia.org/wiki/Guarded_suspension](http://en.wikipedia.org/wiki/Guarded_suspension)
Implementing Guarded Suspension with CVs

- CVs are most often used to implement the *Guarded Suspension* pattern
- Applied to operations requiring both
  - a lock to be acquired &
  - a precondition to be satisfied before the operation can run

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
  cond.await()
doOperationProcessing()
```
Implementing Guarded Suspension with CVs

- e.g., Thread\(_1\) suspends its execution on a CV until Thread\(_n\) notifies it that shared state it's waiting on \textit{may} now be true

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

Note the tentative nature of "may"..
Implementing Guarded Suspension with CVs

- e.g., Thread\(_1\) suspends its execution on a CV until Thread\(_n\) notifies it that shared state it's waiting on *may* now be true.

\[\text{Lock } l = \text{new }\text{Lock}()\]
\[\text{Condition}\ \text{cond} = l.\text{newCondition}()\]

\[\ldots\]
\[l.\text{lock}()\]
\[\text{while (conditionNotSatisfied())}\]
\[\quad \text{cond.await}()\]
\[\quad \text{doOperationProcessing}()\]

- First, a lock must be acquired.
Implementing Guarded Suspension with CVs

- e.g., Thread₁ suspends its execution on a CV until Threadₙ notifies it that shared state it's waiting on may now be true

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

Second, a condition is checked with the lock held..
Implementing Guarded Suspension with CVs

• e.g., Thread₁ suspends its execution on a CV until Threadₙ notifies it that shared state it's waiting on may now be true

```
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

ConditionVariable

- await()
- signal()
- signalAll()

After the condition is satisfied some operation is performed
Implementing Guarded Suspension with CVs

- e.g., Thread$_1$ suspends its execution on a CV until Thread$_n$ notifies it that shared state it's waiting on may now be true

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()

ConditionVariable

- await()
- signal()
- signalAll()
```

However, while the condition is not satisfied the calling thread will block.
Implementing Guarded Suspension with CVs

- e.g., Thread$_1$ suspends its execution on a CV until Thread$_n$ notifies it that shared state it's waiting on may now be true

```
Lock l = new Lock()
Condition cond = l.newCondition()
...

l.lock()
while (conditionNotSatisfied())
cond.await()
doOperationProcessing()
```

Another thread can signal the condition when the shared state may now be true
Implementing Guarded Suspension with CVs

- e.g., Thread₁ suspends its execution on a CV until Threadₙ notifies it that shared state it's waiting on may now be true

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

ConditionVariable
- await()
- signal()
- signalAll()

The lock is reacquired & the condition is rechecked in the loop
Implementing Guarded Suspension with CVs

- A condition can be arbitrarily complex, e.g.
  - A method call, an expression involving shared state, etc.

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```
Implementing Guarded Suspension with CVs

- A condition can be arbitrarily complex, e.g.
  - A method call, an expression involving shared state, etc.
- Any state shared between threads must therefore be protected by a lock associated with the CV

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread *atomically*

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
  cond.await()
doOperationProcessing()

The lock is released when the thread is suspended on the CV
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread *atomically*

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread \textit{atomically}
- Thread$_1$ is suspended until Thread$_n$ signals the CV
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread *atomically*
- Thread₁ is suspended until Threadₙ signals the CV

```
l.lock() cond.signal()
Lock
lock()
unlock()
ConditionVariable
await()
signal()
signalAll()
```
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread *atomically*
- Thread_1 is suspended until Thread_n signals the CV

When a thread is signaled it wakes up & must re-acquire its associated lock
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread *atomically*
- Thread\(_1\) is suspended until Thread\(_n\) signals the CV

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

After the lock is re-acquired the thread can re-evaluate its condition to see if it has become true
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread *atomically*
- Thread$\_1$ is suspended until Thread$\_n$ signals the CV

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()

After the lock is re-acquired & the condition is true then the operation can proceed
Implementing Guarded Suspension with CVs

- Waiting on a CV releases the lock & suspends the thread *atomically*
- Thread$_1$ is suspended until Thread$_n$ signals the CV

```java
Lock l = new Lock()
Condition cond = l.newCondition()
...
l.lock()
while (conditionNotSatisfied())
    cond.await()
doOperationProcessing()
```

However, if the condition is not true then the thread must wait again (which again releases the lock atomically)
End of Java ConditionObject (Part 1)
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.
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
Applying Condition Variables in Practice
Applying Condition Variables in Practice

- CVs are powerful, but can be hard to grok & apply correctly

See en.wikipedia.org/wiki/Grok
Applying Condition Variables in Practice

- CVs are powerful, but can be hard to grok & apply correctly, e.g.
- The protocol for using CVs involves several "moving parts"

[Diagram showing the interaction between threads, objects, and monitors]

CAUTION
BE ALERT!!
MOVING PARTS
Applying Condition Variables in Practice

- CVs are powerful, but can be hard to grok & apply correctly, e.g.
  - The protocol for using CVs involves several “moving parts”
  - i.e., a condition variable & a lock
Applying Condition Variables in Practice

- CVs are powerful, but can be hard to grok & apply correctly, e.g.
  - The protocol for using CVs involves several “moving parts”
  - The non-determinism of concurrency is tricky
Applying Condition Variables in Practice

- CVs are powerful, but can be hard to grok & apply correctly, e.g.
  - The protocol for using CVs involves several “moving parts”
  - The non-determinism of concurrency is tricky
    - i.e., a loop may be needed to ensure a resource is available
Applying Condition Variables in Practice

- CVs are therefore often not used directly by apps, but instead are “hidden” within other abstractions.
Applying Condition Variables in Practice

- CVs are therefore often not used directly by apps, but instead are “hidden” within other abstractions, e.g.
- CVs form the basis for higher-level synchronizers in Java
Applying Condition Variables in Practice

- CVs are therefore often not used directly by apps, but instead are “hidden” within other abstractions, e.g.
  - CVs form the basis for higher-level synchronizers in Java
  - CVs are used in blocking queues & deques in java.util.concurrent* packages

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html
Applying Condition Variables in Practice

- CVs are therefore often not used directly by apps, but instead are “hidden” within other abstractions, e.g.
  - CVs form the basis for higher-level synchronizers in Java
  - CVs are used in blocking queues & deques in java.util.concurrent* packages
    - e.g., ArrayBlockingQueue, which we cover later in this lesson

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html
Applying Condition Variables in Practice

- CVs are therefore often not used directly by apps, but instead are “hidden” within other abstractions, e.g.
  - CVs form the basis for higher-level synchronizers in Java
  - CVs are used in blocking queues & deques in java.util.concurrent* packages
- Java built-in monitor objects

See upcoming lesson on “Java Built-in Monitor Objects”
Applying Condition Variables in Practice

- CVs are therefore often not used directly by apps, but instead are "hidden" within other abstractions, e.g.
  - CVs form the basis for higher-level synchronizers in Java
  - CVs are used in blocking queues & deques in java.util.concurrent packages
  - Java built-in monitor objects
  - The *Monitor Object* pattern

See [www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf](www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf)
Human Known Use of Condition Variables
Human Known Uses of Condition Variables

- A human known use is a pizza delivery protocol
  - Must acquire both the pizza & the keys to deliver the pizza
End of Java ConditionObject (Part 2)
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 ConditionObject
Overview of ConditionObject

- ConditionObject provides the condition variable abstraction

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

**Class AbstractQueuedSynchronizer.ConditionObject**

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

**All Implemented Interfaces:**

- Serializable, Condition

**Enclosing class:**

AbstractQueuedSynchronizer

```java
public class AbstractQueuedSynchronizer.ConditionObject
extends Object
implements Object 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](docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html)
Overview of ConditionObject

- ConditionObject provides the condition variable abstraction
- Implements Condition interface

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

...
Overview of 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 ConditionObject

- A ConditionObject provides a “wait queue” of nodes
Overview of 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
<<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
- acquireSharedNanos(int,long):boolean
- tryAcquireSharedNanos(int,long):boolean
- releaseShared(int):boolean
```
Overview of ConditionObject

- A ConditionObject is *always* used with a lock

See earlier part on “Java ReentrantLock”
Overview of ConditionObject

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

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

newCondition() returns a ConditionObject that can be used with this lock

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html#newCondition---
Overview of ConditionObject

- Both ReentrantLock & ConditionObject have internal queues

Diagram:

- Consumer
- ArrayBlockingQueue
  - put()
  - take()
- Producer
- ConditionObject
  - await()
  - signal()
  - signalAll()
- ReentrantLock
  - lock()
  - unlock()
  - newCondition()
Overview of ConditionObject

- Both ReentrantLock & ConditionObject have internal queues

**Diagram:**

- **Reentrant Lock**
  - lock()
  - unlock()
  - newCondition()

- **ConditionObject**
  - await()
  - signal()
  - signalAll()

- **ArrayBlocking Queue**
  - put()
  - take()

- **Producer**
  - put()

- **Consumer**
  - take()

**Note:**

Queues up threads that want to acquire the lock
Overview of ConditionObject

- Both ReentrantLock & ConditionObject have internal queues

Queues up threads that are waiting for some condition to become true

```
Consumer

ArrayBlockingQueue
  put()
  take()

Producer

ConditionObject
  await()
  signal()
  signalAll()

ReentrantLock
  lock()
  unlock()
  newCondition()
```
Overview of ConditionObject

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

Two ConditionObjects: mNotEmpty & mNotFull
Overview of ConditionObject

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

![Diagram of ConditionObject interactions]

- Consumer
- ArrayBlockingQueue
  - put()
  - take()
- Producer
- ConditionObject
  - await()
  - signal()
  - signalAll()
- ReentrantLock
  - lock()
  - unlock()
  - newCondition()
Overview of 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 ConditionObject

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

**Diagram:**
- **Producer**
  - `put()`
  - `<<contains>>`
- **Simple BlockingQueue**
  - `put()`
  - `take()`
  - `<<contains>>`
- **Consumer**
  - `take()`
  - `<<contains>>`
- **Wait Queue**
  - `wait()`
  - `notify()`
  - `notifyAll()`
  - `1`
  - `uses` **Entrance Queue**

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

See upcoming lesson on “Java Built-in Monitor Objects”
• 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
In contrast, Java’s built-in monitor objects only support one monitor condition. It yields inefficient programs that require excessive notifications & use of notifyAll().

Overview of ConditionObject

- Must wake up both producers & consumers on every change to the queue, even if it cannot possibly allow given thread to proceed.

See stackoverflow.com/questions/18490636/condition-give-the-effect-of-having-multiple-wait-sets-per-object
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()
    { ... }

    ...

These methods are implemented via the AbstractQueuedSynchronizer framework
```
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()
    { ... }

    ...  

See upcoming lesson on “Java Built-in Monitor Objects”
```
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()
    throws InterruptedException
  {
    ...  }

  ...  

  ...  
```
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 ConditionObject to the queue for the owning lock

```java
public class ConditionObject
    implements Condition,
    java.io.Serializable {
    ...
    /** Wakeup longest waiting thread. */
    public final void 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 ConditionObject 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 {
    ...
    /** Wakeup all waiting threads. */
    public final void 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 ConditionObject 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!!

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

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 Type</th>
<th>Method Name</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>
• 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>

Unlike Java’s built-in monitor object timed `wait()` calls, these timed `await*()` calls gives a sensible return value.
End of Java ConditionObject
(Part 3)