| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/Condition.html) | [**Tree**](http://docs.google.com/package-tree.html) | [**Deprecated**](http://docs.google.com/deprecated-list.html) | [**Index**](http://docs.google.com/index-files/index-1.html) | [**Help**](http://docs.google.com/help-doc.html) | | --- | --- | --- | --- | --- | --- | --- | --- | | | ***Java™ Platform***  ***Standard Ed. 6*** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**PREV CLASS**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html)   [**NEXT CLASS**](http://docs.google.com/java/util/concurrent/locks/Lock.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/locks/Condition.html)    [**NO FRAMES**](http://docs.google.com/Condition.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |
| SUMMARY: NESTED | FIELD | CONSTR | [METHOD](#3znysh7) | DETAIL: FIELD | CONSTR | [METHOD](#2et92p0) |

## **java.util.concurrent.locks**

Interface Condition

**All Known Implementing Classes:** [AbstractQueuedLongSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedLongSynchronizer.ConditionObject.html), [AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html)

public interface **Condition**

Condition factors out the Object monitor methods ([wait](http://docs.google.com/java/lang/Object.html#wait()), [notify](http://docs.google.com/java/lang/Object.html#notify()) and [notifyAll](http://docs.google.com/java/lang/Object.html#notifyAll())) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary [Lock](http://docs.google.com/java/util/concurrent/locks/Lock.html) 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](http://docs.google.com/java/util/concurrent/locks/Lock.html) instance use its [newCondition()](http://docs.google.com/java/util/concurrent/locks/Lock.html#newCondition()) method.

As an example, suppose we have a bounded buffer which supports put and take methods. If a take is attempted on an empty buffer, then the thread will block until an item becomes available; if a put is attempted on a full buffer, then the thread will block until a space becomes available. We would like to keep waiting put threads and take threads in separate wait-sets so that we can use the optimization of only notifying a single thread at a time when items or spaces become available in the buffer. This can be achieved using two [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) instances.

class BoundedBuffer {  
 **final Lock lock = new ReentrantLock();**  
 final Condition notFull = **lock.newCondition();**   
 final Condition notEmpty = **lock.newCondition();**   
  
 final Object[] items = new Object[100];  
 int putptr, takeptr, count;  
  
 public void put(Object x) throws InterruptedException {  
 **lock.lock();  
 try {**  
 while (count == items.length)  
 **notFull.await();**  
 items[putptr] = x;  
 if (++putptr == items.length) putptr = 0;  
 ++count;  
 **notEmpty.signal();**  
 **} finally {  
 lock.unlock();  
 }**  
 }  
  
 public Object take() throws InterruptedException {  
 **lock.lock();  
 try {**  
 while (count == 0)  
 **notEmpty.await();**  
 Object x = items[takeptr];  
 if (++takeptr == items.length) takeptr = 0;  
 --count;  
 **notFull.signal();**  
 return x;  
 **} finally {  
 lock.unlock();  
 }**  
 }  
 }

(The [ArrayBlockingQueue](http://docs.google.com/java/util/concurrent/ArrayBlockingQueue.html) class provides this functionality, so there is no reason to implement this sample usage class.)

A Condition implementation can provide behavior and semantics that is different from that of the Object monitor methods, such as guaranteed ordering for notifications, or not requiring a lock to be held when performing notifications. If an implementation provides such specialized semantics then the implementation must document those semantics.

Note that Condition instances are just normal objects and can themselves be used as the target in a synchronized statement, and can have their own monitor [wait](http://docs.google.com/java/lang/Object.html#wait(long)) and [notification](http://docs.google.com/java/lang/Object.html#notify()) methods invoked. Acquiring the monitor lock of a Condition instance, or using its monitor methods, has no specified relationship with acquiring the [Lock](http://docs.google.com/java/util/concurrent/locks/Lock.html) associated with that Condition or the use of its [waiting](http://docs.google.com/java/util/concurrent/locks/Condition.html#await()) and [signalling](http://docs.google.com/java/util/concurrent/locks/Condition.html#signal()) methods. It is recommended that to avoid confusion you never use Condition instances in this way, except perhaps within their own implementation.

Except where noted, passing a null value for any parameter will result in a [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) being thrown.

### Implementation Considerations

When waiting upon a Condition, a "*spurious wakeup*" is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for. An implementation is free to remove the possibility of spurious wakeups but it is recommended that applications programmers always assume that they can occur and so always wait in a loop.

The three forms of condition waiting (interruptible, non-interruptible, and timed) may differ in their ease of implementation on some platforms and in their performance characteristics. In particular, it may be difficult to provide these features and maintain specific semantics such as ordering guarantees. Further, the ability to interrupt the actual suspension of the thread may not always be feasible to implement on all platforms.

Consequently, an implementation is not required to define exactly the same guarantees or semantics for all three forms of waiting, nor is it required to support interruption of the actual suspension of the thread.

An implementation is required to clearly document the semantics and guarantees provided by each of the waiting methods, and when an implementation does support interruption of thread suspension then it must obey the interruption semantics as defined in this interface.

As interruption generally implies cancellation, and checks for interruption are often infrequent, an implementation can favor responding to an interrupt over normal method return. This is true even if it can be shown that the interrupt occurred after another action may have unblocked the thread. An implementation should document this behavior.

**Since:** 1.5

| **Method Summary** | |
| --- | --- |
| void | [**await**](http://docs.google.com/java/util/concurrent/locks/Condition.html#await())()            Causes the current thread to wait until it is signalled or [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()). |
| boolean | [**await**](http://docs.google.com/java/util/concurrent/locks/Condition.html#await(long,%20java.util.concurrent.TimeUnit))(long time, [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit)            Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses. |
| long | [**awaitNanos**](http://docs.google.com/java/util/concurrent/locks/Condition.html#awaitNanos(long))(long nanosTimeout)            Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses. |
| void | [**awaitUninterruptibly**](http://docs.google.com/java/util/concurrent/locks/Condition.html#awaitUninterruptibly())()            Causes the current thread to wait until it is signalled. |
| boolean | [**awaitUntil**](http://docs.google.com/java/util/concurrent/locks/Condition.html#awaitUntil(java.util.Date))([Date](http://docs.google.com/java/util/Date.html) deadline)            Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses. |
| void | [**signal**](http://docs.google.com/java/util/concurrent/locks/Condition.html#signal())()            Wakes up one waiting thread. |
| void | [**signalAll**](http://docs.google.com/java/util/concurrent/locks/Condition.html#signalAll())()            Wakes up all waiting threads. |

| **Method Detail** |
| --- |

### await

void **await**()  
 throws [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html)

Causes the current thread to wait until it is signalled or [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()).

The lock associated with this Condition is atomically released and the current thread becomes disabled for thread scheduling purposes and lies dormant until *one* of four things happens:

* Some other thread invokes the [signal()](http://docs.google.com/java/util/concurrent/locks/Condition.html#signal()) method for this Condition and the current thread happens to be chosen as the thread to be awakened; or
* Some other thread invokes the [signalAll()](http://docs.google.com/java/util/concurrent/locks/Condition.html#signalAll()) method for this Condition; or
* Some other thread [interrupts](http://docs.google.com/java/lang/Thread.html#interrupt()) the current thread, and interruption of thread suspension is supported; or
* A "*spurious wakeup*" occurs.

In all cases, before this method can return the current thread must re-acquire the lock associated with this condition. When the thread returns it is *guaranteed* to hold this lock.

If the current thread:

* has its interrupted status set on entry to this method; or
* is [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()) while waiting and interruption of thread suspension is supported,

then [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) is thrown and the current thread's interrupted status is cleared. It is not specified, in the first case, whether or not the test for interruption occurs before the lock is released.

**Implementation Considerations**

The current thread is assumed to hold the lock associated with this Condition when this method is called. It is up to the implementation to determine if this is the case and if not, how to respond. Typically, an exception will be thrown (such as [IllegalMonitorStateException](http://docs.google.com/java/lang/IllegalMonitorStateException.html)) and the implementation must document that fact.

An implementation can favor responding to an interrupt over normal method return in response to a signal. In that case the implementation must ensure that the signal is redirected to another waiting thread, if there is one.

**Throws:** [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) - if the current thread is interrupted (and interruption of thread suspension is supported)

### awaitUninterruptibly

void **awaitUninterruptibly**()

Causes the current thread to wait until it is signalled.

The lock associated with this condition is atomically released and the current thread becomes disabled for thread scheduling purposes and lies dormant until *one* of three things happens:

* Some other thread invokes the [signal()](http://docs.google.com/java/util/concurrent/locks/Condition.html#signal()) method for this Condition and the current thread happens to be chosen as the thread to be awakened; or
* Some other thread invokes the [signalAll()](http://docs.google.com/java/util/concurrent/locks/Condition.html#signalAll()) method for this Condition; or
* A "*spurious wakeup*" occurs.

In all cases, before this method can return the current thread must re-acquire the lock associated with this condition. When the thread returns it is *guaranteed* to hold this lock.

If the current thread's interrupted status is set when it enters this method, or it is [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()) while waiting, it will continue to wait until signalled. When it finally returns from this method its interrupted status will still be set.

**Implementation Considerations**

The current thread is assumed to hold the lock associated with this Condition when this method is called. It is up to the implementation to determine if this is the case and if not, how to respond. Typically, an exception will be thrown (such as [IllegalMonitorStateException](http://docs.google.com/java/lang/IllegalMonitorStateException.html)) and the implementation must document that fact.

### awaitNanos

long **awaitNanos**(long nanosTimeout)  
 throws [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html)

Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.

The lock associated with this condition is atomically released and the current thread becomes disabled for thread scheduling purposes and lies dormant until *one* of five things happens:

* Some other thread invokes the [signal()](http://docs.google.com/java/util/concurrent/locks/Condition.html#signal()) method for this Condition and the current thread happens to be chosen as the thread to be awakened; or
* Some other thread invokes the [signalAll()](http://docs.google.com/java/util/concurrent/locks/Condition.html#signalAll()) method for this Condition; or
* Some other thread [interrupts](http://docs.google.com/java/lang/Thread.html#interrupt()) the current thread, and interruption of thread suspension is supported; or
* The specified waiting time elapses; or
* A "*spurious wakeup*" occurs.

In all cases, before this method can return the current thread must re-acquire the lock associated with this condition. When the thread returns it is *guaranteed* to hold this lock.

If the current thread:

* has its interrupted status set on entry to this method; or
* is [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()) while waiting and interruption of thread suspension is supported,

then [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) is thrown and the current thread's interrupted status is cleared. It is not specified, in the first case, whether or not the test for interruption occurs before the lock is released.

The method returns an estimate of the number of nanoseconds remaining to wait given the supplied nanosTimeout value upon return, or a value less than or equal to zero if it timed out. This value can be used to determine whether and how long to re-wait in cases where the wait returns but an awaited condition still does not hold. Typical uses of this method take the following form:

synchronized boolean aMethod(long timeout, TimeUnit unit) {  
 long nanosTimeout = unit.toNanos(timeout);  
 while (!conditionBeingWaitedFor) {  
 if (nanosTimeout > 0)  
 nanosTimeout = theCondition.awaitNanos(nanosTimeout);  
 else  
 return false;  
 }  
 // ...  
 }

Design note: This method requires a nanosecond argument so as to avoid truncation errors in reporting remaining times. Such precision loss would make it difficult for programmers to ensure that total waiting times are not systematically shorter than specified when re-waits occur.

**Implementation Considerations**

The current thread is assumed to hold the lock associated with this Condition when this method is called. It is up to the implementation to determine if this is the case and if not, how to respond. Typically, an exception will be thrown (such as [IllegalMonitorStateException](http://docs.google.com/java/lang/IllegalMonitorStateException.html)) and the implementation must document that fact.

An implementation can favor responding to an interrupt over normal method return in response to a signal, or over indicating the elapse of the specified waiting time. In either case the implementation must ensure that the signal is redirected to another waiting thread, if there is one.

**Parameters:**nanosTimeout - the maximum time to wait, in nanoseconds **Returns:**an estimate of the nanosTimeout value minus the time spent waiting upon return from this method. A positive value may be used as the argument to a subsequent call to this method to finish waiting out the desired time. A value less than or equal to zero indicates that no time remains. **Throws:** [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) - if the current thread is interrupted (and interruption of thread suspension is supported)

### await

boolean **await**(long time,  
 [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit)  
 throws [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html)

Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses. This method is behaviorally equivalent to:

awaitNanos(unit.toNanos(time)) > 0

**Parameters:**time - the maximum time to waitunit - the time unit of the time argument **Returns:**false if the waiting time detectably elapsed before return from the method, else true **Throws:** [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) - if the current thread is interrupted (and interruption of thread suspension is supported)

### awaitUntil

boolean **awaitUntil**([Date](http://docs.google.com/java/util/Date.html) deadline)  
 throws [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html)

Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses.

The lock associated with this condition is atomically released and the current thread becomes disabled for thread scheduling purposes and lies dormant until *one* of five things happens:

* Some other thread invokes the [signal()](http://docs.google.com/java/util/concurrent/locks/Condition.html#signal()) method for this Condition and the current thread happens to be chosen as the thread to be awakened; or
* Some other thread invokes the [signalAll()](http://docs.google.com/java/util/concurrent/locks/Condition.html#signalAll()) method for this Condition; or
* Some other thread [interrupts](http://docs.google.com/java/lang/Thread.html#interrupt()) the current thread, and interruption of thread suspension is supported; or
* The specified deadline elapses; or
* A "*spurious wakeup*" occurs.

In all cases, before this method can return the current thread must re-acquire the lock associated with this condition. When the thread returns it is *guaranteed* to hold this lock.

If the current thread:

* has its interrupted status set on entry to this method; or
* is [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()) while waiting and interruption of thread suspension is supported,

then [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) is thrown and the current thread's interrupted status is cleared. It is not specified, in the first case, whether or not the test for interruption occurs before the lock is released.

The return value indicates whether the deadline has elapsed, which can be used as follows:

synchronized boolean aMethod(Date deadline) {  
 boolean stillWaiting = true;  
 while (!conditionBeingWaitedFor) {  
 if (stillWaiting)  
 stillWaiting = theCondition.awaitUntil(deadline);  
 else  
 return false;  
 }  
 // ...  
 }

**Implementation Considerations**

The current thread is assumed to hold the lock associated with this Condition when this method is called. It is up to the implementation to determine if this is the case and if not, how to respond. Typically, an exception will be thrown (such as [IllegalMonitorStateException](http://docs.google.com/java/lang/IllegalMonitorStateException.html)) and the implementation must document that fact.

An implementation can favor responding to an interrupt over normal method return in response to a signal, or over indicating the passing of the specified deadline. In either case the implementation must ensure that the signal is redirected to another waiting thread, if there is one.

**Parameters:**deadline - the absolute time to wait until **Returns:**false if the deadline has elapsed upon return, else true **Throws:** [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) - if the current thread is interrupted (and interruption of thread suspension is supported)

### signal

void **signal**()

Wakes up one waiting thread.

If any threads are waiting on this condition then one is selected for waking up. That thread must then re-acquire the lock before returning from await.

### signalAll

void **signalAll**()

Wakes up all waiting threads.

If any threads are waiting on this condition then they are all woken up. Each thread must re-acquire the lock before it can return from await.

| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/Condition.html) | [**Tree**](http://docs.google.com/package-tree.html) | [**Deprecated**](http://docs.google.com/deprecated-list.html) | [**Index**](http://docs.google.com/index-files/index-1.html) | [**Help**](http://docs.google.com/help-doc.html) | | --- | --- | --- | --- | --- | --- | --- | --- | | | ***Java™ Platform***  ***Standard Ed. 6*** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**PREV CLASS**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html)   [**NEXT CLASS**](http://docs.google.com/java/util/concurrent/locks/Lock.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/locks/Condition.html)    [**NO FRAMES**](http://docs.google.com/Condition.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |
| SUMMARY: NESTED | FIELD | CONSTR | [METHOD](#3znysh7) | DETAIL: FIELD | CONSTR | [METHOD](#2et92p0) |

[Submit a bug or feature](http://bugs.sun.com/services/bugreport/index.jsp)

For further API reference and developer documentation, see [Java SE Developer Documentation](http://docs.google.com/webnotes/devdocs-vs-specs.html). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2006 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](http://docs.google.com/legal/license.html). Also see the [documentation redistribution policy](http://java.sun.com/docs/redist.html).