| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/AbstractQueuedSynchronizer.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/AbstractQueuedLongSynchronizer.ConditionObject.html)   [**NEXT CLASS**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/locks/AbstractQueuedSynchronizer.html)    [**NO FRAMES**](http://docs.google.com/AbstractQueuedSynchronizer.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |
| SUMMARY: [NESTED](#3znysh7) | FIELD | [CONSTR](#2et92p0) | [METHOD](#tyjcwt) | DETAIL: FIELD | [CONSTR](#4d34og8) | [METHOD](#17dp8vu) |

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

Class AbstractQueuedSynchronizer

[java.lang.Object](http://docs.google.com/java/lang/Object.html)  
 [java.util.concurrent.locks.AbstractOwnableSynchronizer](http://docs.google.com/java/util/concurrent/locks/AbstractOwnableSynchronizer.html)  
 **java.util.concurrent.locks.AbstractQueuedSynchronizer**

**All Implemented Interfaces:** [Serializable](http://docs.google.com/java/io/Serializable.html)

public abstract class **AbstractQueuedSynchronizer**extends [AbstractOwnableSynchronizer](http://docs.google.com/java/util/concurrent/locks/AbstractOwnableSynchronizer.html)implements [Serializable](http://docs.google.com/java/io/Serializable.html)

Provides a framework for implementing blocking locks and related synchronizers (semaphores, events, etc) that rely on first-in-first-out (FIFO) wait queues. This class is designed to be a useful basis for most kinds of synchronizers that rely on a single atomic int value to represent state. Subclasses must define the protected methods that change this state, and which define what that state means in terms of this object being acquired or released. Given these, the other methods in this class carry out all queuing and blocking mechanics. Subclasses can maintain other state fields, but only the atomically updated int value manipulated using methods [getState()](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getState()), [setState(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#setState(int)) and [compareAndSetState(int, int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#compareAndSetState(int,%20int)) is tracked with respect to synchronization.

Subclasses should be defined as non-public internal helper classes that are used to implement the synchronization properties of their enclosing class. Class AbstractQueuedSynchronizer does not implement any synchronization interface. Instead it defines methods such as [acquireInterruptibly(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#acquireInterruptibly(int)) that can be invoked as appropriate by concrete locks and related synchronizers to implement their public methods.

This class supports either or both a default *exclusive* mode and a *shared* mode. When acquired in exclusive mode, attempted acquires by other threads cannot succeed. Shared mode acquires by multiple threads may (but need not) succeed. This class does not "understand" these differences except in the mechanical sense that when a shared mode acquire succeeds, the next waiting thread (if one exists) must also determine whether it can acquire as well. Threads waiting in the different modes share the same FIFO queue. Usually, implementation subclasses support only one of these modes, but both can come into play for example in a [ReadWriteLock](http://docs.google.com/java/util/concurrent/locks/ReadWriteLock.html). Subclasses that support only exclusive or only shared modes need not define the methods supporting the unused mode.

This class defines a nested [AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) class that can be used as a [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) implementation by subclasses supporting exclusive mode for which method [isHeldExclusively()](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#isHeldExclusively()) reports whether synchronization is exclusively held with respect to the current thread, method [release(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#release(int)) invoked with the current [getState()](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getState()) value fully releases this object, and [acquire(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#acquire(int)), given this saved state value, eventually restores this object to its previous acquired state. No AbstractQueuedSynchronizer method otherwise creates such a condition, so if this constraint cannot be met, do not use it. The behavior of [AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) depends of course on the semantics of its synchronizer implementation.

This class provides inspection, instrumentation, and monitoring methods for the internal queue, as well as similar methods for condition objects. These can be exported as desired into classes using an AbstractQueuedSynchronizer for their synchronization mechanics.

Serialization of this class stores only the underlying atomic integer maintaining state, so deserialized objects have empty thread queues. Typical subclasses requiring serializability will define a readObject method that restores this to a known initial state upon deserialization.

### Usage

To use this class as the basis of a synchronizer, redefine the following methods, as applicable, by inspecting and/or modifying the synchronization state using [getState()](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getState()), [setState(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#setState(int)) and/or [compareAndSetState(int, int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#compareAndSetState(int,%20int)):

* [tryAcquire(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquire(int))
* [tryRelease(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryRelease(int))
* [tryAcquireShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireShared(int))
* [tryReleaseShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryReleaseShared(int))
* [isHeldExclusively()](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#isHeldExclusively())

Each of these methods by default throws [UnsupportedOperationException](http://docs.google.com/java/lang/UnsupportedOperationException.html). Implementations of these methods must be internally thread-safe, and should in general be short and not block. Defining these methods is the *only* supported means of using this class. All other methods are declared final because they cannot be independently varied.

You may also find the inherited methods from [AbstractOwnableSynchronizer](http://docs.google.com/java/util/concurrent/locks/AbstractOwnableSynchronizer.html) useful to keep track of the thread owning an exclusive synchronizer. You are encouraged to use them -- this enables monitoring and diagnostic tools to assist users in determining which threads hold locks.

Even though this class is based on an internal FIFO queue, it does not automatically enforce FIFO acquisition policies. The core of exclusive synchronization takes the form:

Acquire:  
 while (!tryAcquire(arg)) {  
 *enqueue thread if it is not already queued*;  
 *possibly block current thread*;  
 }  
  
 Release:  
 if (tryRelease(arg))  
 *unblock the first queued thread*;

(Shared mode is similar but may involve cascading signals.)

Because checks in acquire are invoked before enqueuing, a newly acquiring thread may *barge* ahead of others that are blocked and queued. However, you can, if desired, define tryAcquire and/or tryAcquireShared to disable barging by internally invoking one or more of the inspection methods. In particular, a strict FIFO lock can define tryAcquire to immediately return false if [getFirstQueuedThread()](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getFirstQueuedThread()) does not return the current thread. A normally preferable non-strict fair version can immediately return false only if [hasQueuedThreads()](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#hasQueuedThreads()) returns true and getFirstQueuedThread is not the current thread; or equivalently, that getFirstQueuedThread is both non-null and not the current thread. Further variations are possible.

Throughput and scalability are generally highest for the default barging (also known as *greedy*, *renouncement*, and *convoy-avoidance*) strategy. While this is not guaranteed to be fair or starvation-free, earlier queued threads are allowed to recontend before later queued threads, and each recontention has an unbiased chance to succeed against incoming threads. Also, while acquires do not "spin" in the usual sense, they may perform multiple invocations of tryAcquire interspersed with other computations before blocking. This gives most of the benefits of spins when exclusive synchronization is only briefly held, without most of the liabilities when it isn't. If so desired, you can augment this by preceding calls to acquire methods with "fast-path" checks, possibly prechecking [hasContended()](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#hasContended()) and/or [hasQueuedThreads()](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#hasQueuedThreads()) to only do so if the synchronizer is likely not to be contended.

This class provides an efficient and scalable basis for synchronization in part by specializing its range of use to synchronizers that can rely on int state, acquire, and release parameters, and an internal FIFO wait queue. When this does not suffice, you can build synchronizers from a lower level using [atomic](http://docs.google.com/java/util/concurrent/atomic/package-summary.html) classes, your own custom [Queue](http://docs.google.com/java/util/Queue.html) classes, and [LockSupport](http://docs.google.com/java/util/concurrent/locks/LockSupport.html) blocking support.

### Usage Examples

Here is a non-reentrant mutual exclusion lock class that uses the value zero to represent the unlocked state, and one to represent the locked state. While a non-reentrant lock does not strictly require recording of the current owner thread, this class does so anyway to make usage easier to monitor. It also supports conditions and exposes one of the instrumentation methods:

class Mutex implements Lock, java.io.Serializable {  
  
 // Our internal helper class  
 private static class Sync extends AbstractQueuedSynchronizer {  
 // Report whether in locked state  
 protected boolean isHeldExclusively() {  
 return getState() == 1;  
 }  
  
 // Acquire the lock if state is zero  
 public boolean tryAcquire(int acquires) {  
 assert acquires == 1; // Otherwise unused  
 if (compareAndSetState(0, 1)) {  
 setExclusiveOwnerThread(Thread.currentThread());  
 return true;  
 }  
 return false;  
 }  
  
 // Release the lock by setting state to zero  
 protected boolean tryRelease(int releases) {  
 assert releases == 1; // Otherwise unused  
 if (getState() == 0) throw new IllegalMonitorStateException();  
 setExclusiveOwnerThread(null);  
 setState(0);  
 return true;  
 }  
  
 // Provide a Condition  
 Condition newCondition() { return new ConditionObject(); }  
  
 // Deserialize properly  
 private void readObject(ObjectInputStream s)  
 throws IOException, ClassNotFoundException {  
 s.defaultReadObject();  
 setState(0); // reset to unlocked state  
 }  
 }  
  
 // The sync object does all the hard work. We just forward to it.  
 private final Sync sync = new Sync();  
  
 public void lock() { sync.acquire(1); }  
 public boolean tryLock() { return sync.tryAcquire(1); }  
 public void unlock() { sync.release(1); }  
 public Condition newCondition() { return sync.newCondition(); }  
 public boolean isLocked() { return sync.isHeldExclusively(); }  
 public boolean hasQueuedThreads() { return sync.hasQueuedThreads(); }  
 public void lockInterruptibly() throws InterruptedException {  
 sync.acquireInterruptibly(1);  
 }  
 public boolean tryLock(long timeout, TimeUnit unit)  
 throws InterruptedException {  
 return sync.tryAcquireNanos(1, unit.toNanos(timeout));  
 }  
 }

Here is a latch class that is like a [CountDownLatch](http://docs.google.com/java/util/concurrent/CountDownLatch.html) except that it only requires a single signal to fire. Because a latch is non-exclusive, it uses the shared acquire and release methods.

class BooleanLatch {  
  
 private static class Sync extends AbstractQueuedSynchronizer {  
 boolean isSignalled() { return getState() != 0; }  
  
 protected int tryAcquireShared(int ignore) {  
 return isSignalled()? 1 : -1;  
 }  
  
 protected boolean tryReleaseShared(int ignore) {  
 setState(1);  
 return true;  
 }  
 }  
  
 private final Sync sync = new Sync();  
 public boolean isSignalled() { return sync.isSignalled(); }  
 public void signal() { sync.releaseShared(1); }  
 public void await() throws InterruptedException {  
 sync.acquireSharedInterruptibly(1);  
 }  
 }

**Since:** 1.5 **See Also:**[Serialized Form](http://docs.google.com/serialized-form.html#java.util.concurrent.locks.AbstractQueuedSynchronizer)

| **Nested Class Summary** | |
| --- | --- |
| class | [**AbstractQueuedSynchronizer.ConditionObject**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html)            Condition implementation for a [AbstractQueuedSynchronizer](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html) serving as the basis of a [Lock](http://docs.google.com/java/util/concurrent/locks/Lock.html) implementation. |

| **Constructor Summary** | |
| --- | --- |
| protected | [**AbstractQueuedSynchronizer**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#AbstractQueuedSynchronizer())()            Creates a new AbstractQueuedSynchronizer instance with initial synchronization state of zero. |

| **Method Summary** | |
| --- | --- |
| void | [**acquire**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#acquire(int))(int arg)            Acquires in exclusive mode, ignoring interrupts. |
| void | [**acquireInterruptibly**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#acquireInterruptibly(int))(int arg)            Acquires in exclusive mode, aborting if interrupted. |
| void | [**acquireShared**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#acquireShared(int))(int arg)            Acquires in shared mode, ignoring interrupts. |
| void | [**acquireSharedInterruptibly**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#acquireSharedInterruptibly(int))(int arg)            Acquires in shared mode, aborting if interrupted. |
| protected  boolean | [**compareAndSetState**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#compareAndSetState(int,%20int))(int expect, int update)            Atomically sets synchronization state to the given updated value if the current state value equals the expected value. |
| [Collection](http://docs.google.com/java/util/Collection.html)<[Thread](http://docs.google.com/java/lang/Thread.html)> | [**getExclusiveQueuedThreads**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getExclusiveQueuedThreads())()            Returns a collection containing threads that may be waiting to acquire in exclusive mode. |
| [Thread](http://docs.google.com/java/lang/Thread.html) | [**getFirstQueuedThread**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getFirstQueuedThread())()            Returns the first (longest-waiting) thread in the queue, or null if no threads are currently queued. |
| [Collection](http://docs.google.com/java/util/Collection.html)<[Thread](http://docs.google.com/java/lang/Thread.html)> | [**getQueuedThreads**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getQueuedThreads())()            Returns a collection containing threads that may be waiting to acquire. |
| int | [**getQueueLength**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getQueueLength())()            Returns an estimate of the number of threads waiting to acquire. |
| [Collection](http://docs.google.com/java/util/Collection.html)<[Thread](http://docs.google.com/java/lang/Thread.html)> | [**getSharedQueuedThreads**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getSharedQueuedThreads())()            Returns a collection containing threads that may be waiting to acquire in shared mode. |
| protected  int | [**getState**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getState())()            Returns the current value of synchronization state. |
| [Collection](http://docs.google.com/java/util/Collection.html)<[Thread](http://docs.google.com/java/lang/Thread.html)> | [**getWaitingThreads**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getWaitingThreads(java.util.concurrent.locks.AbstractQueuedSynchronizer.ConditionObject))([AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) condition)            Returns a collection containing those threads that may be waiting on the given condition associated with this synchronizer. |
| int | [**getWaitQueueLength**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getWaitQueueLength(java.util.concurrent.locks.AbstractQueuedSynchronizer.ConditionObject))([AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) condition)            Returns an estimate of the number of threads waiting on the given condition associated with this synchronizer. |
| boolean | [**hasContended**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#hasContended())()            Queries whether any threads have ever contended to acquire this synchronizer; that is if an acquire method has ever blocked. |
| boolean | [**hasQueuedThreads**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#hasQueuedThreads())()            Queries whether any threads are waiting to acquire. |
| boolean | [**hasWaiters**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#hasWaiters(java.util.concurrent.locks.AbstractQueuedSynchronizer.ConditionObject))([AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) condition)            Queries whether any threads are waiting on the given condition associated with this synchronizer. |
| protected  boolean | [**isHeldExclusively**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#isHeldExclusively())()            Returns true if synchronization is held exclusively with respect to the current (calling) thread. |
| boolean | [**isQueued**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#isQueued(java.lang.Thread))([Thread](http://docs.google.com/java/lang/Thread.html) thread)            Returns true if the given thread is currently queued. |
| boolean | [**owns**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#owns(java.util.concurrent.locks.AbstractQueuedSynchronizer.ConditionObject))([AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) condition)            Queries whether the given ConditionObject uses this synchronizer as its lock. |
| boolean | [**release**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#release(int))(int arg)            Releases in exclusive mode. |
| boolean | [**releaseShared**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#releaseShared(int))(int arg)            Releases in shared mode. |
| protected  void | [**setState**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#setState(int))(int newState)            Sets the value of synchronization state. |
| [String](http://docs.google.com/java/lang/String.html) | [**toString**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#toString())()            Returns a string identifying this synchronizer, as well as its state. |
| protected  boolean | [**tryAcquire**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquire(int))(int arg)            Attempts to acquire in exclusive mode. |
| boolean | [**tryAcquireNanos**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireNanos(int,%20long))(int arg, long nanosTimeout)            Attempts to acquire in exclusive mode, aborting if interrupted, and failing if the given timeout elapses. |
| protected  int | [**tryAcquireShared**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireShared(int))(int arg)            Attempts to acquire in shared mode. |
| boolean | [**tryAcquireSharedNanos**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireSharedNanos(int,%20long))(int arg, long nanosTimeout)            Attempts to acquire in shared mode, aborting if interrupted, and failing if the given timeout elapses. |
| protected  boolean | [**tryRelease**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryRelease(int))(int arg)            Attempts to set the state to reflect a release in exclusive mode. |
| protected  boolean | [**tryReleaseShared**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryReleaseShared(int))(int arg)            Attempts to set the state to reflect a release in shared mode. |

| **Methods inherited from class java.util.concurrent.locks.**[**AbstractOwnableSynchronizer**](http://docs.google.com/java/util/concurrent/locks/AbstractOwnableSynchronizer.html) |
| --- |
| [getExclusiveOwnerThread](http://docs.google.com/java/util/concurrent/locks/AbstractOwnableSynchronizer.html#getExclusiveOwnerThread()), [setExclusiveOwnerThread](http://docs.google.com/java/util/concurrent/locks/AbstractOwnableSynchronizer.html#setExclusiveOwnerThread(java.lang.Thread)) |

| **Methods inherited from class java.lang.**[**Object**](http://docs.google.com/java/lang/Object.html) |
| --- |
| [clone](http://docs.google.com/java/lang/Object.html#clone()), [equals](http://docs.google.com/java/lang/Object.html#equals(java.lang.Object)), [finalize](http://docs.google.com/java/lang/Object.html#finalize()), [getClass](http://docs.google.com/java/lang/Object.html#getClass()), [hashCode](http://docs.google.com/java/lang/Object.html#hashCode()), [notify](http://docs.google.com/java/lang/Object.html#notify()), [notifyAll](http://docs.google.com/java/lang/Object.html#notifyAll()), [wait](http://docs.google.com/java/lang/Object.html#wait()), [wait](http://docs.google.com/java/lang/Object.html#wait(long)), [wait](http://docs.google.com/java/lang/Object.html#wait(long,%20int)) |

| **Constructor Detail** |
| --- |

### AbstractQueuedSynchronizer

protected **AbstractQueuedSynchronizer**()

Creates a new AbstractQueuedSynchronizer instance with initial synchronization state of zero.

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

### getState

protected final int **getState**()

Returns the current value of synchronization state. This operation has memory semantics of a volatile read.

**Returns:**current state value

### setState

protected final void **setState**(int newState)

Sets the value of synchronization state. This operation has memory semantics of a volatile write.

**Parameters:**newState - the new state value

### compareAndSetState

protected final boolean **compareAndSetState**(int expect,  
 int update)

Atomically sets synchronization state to the given updated value if the current state value equals the expected value. This operation has memory semantics of a volatile read and write.

**Parameters:**expect - the expected valueupdate - the new value **Returns:**true if successful. False return indicates that the actual value was not equal to the expected value.

### tryAcquire

protected boolean **tryAcquire**(int arg)

Attempts to acquire in exclusive mode. This method should query if the state of the object permits it to be acquired in the exclusive mode, and if so to acquire it.

This method is always invoked by the thread performing acquire. If this method reports failure, the acquire method may queue the thread, if it is not already queued, until it is signalled by a release from some other thread. This can be used to implement method [Lock.tryLock()](http://docs.google.com/java/util/concurrent/locks/Lock.html#tryLock()).

The default implementation throws [UnsupportedOperationException](http://docs.google.com/java/lang/UnsupportedOperationException.html).

**Parameters:**arg - the acquire argument. This value is always the one passed to an acquire method, or is the value saved on entry to a condition wait. The value is otherwise uninterpreted and can represent anything you like. **Returns:**true if successful. Upon success, this object has been acquired. **Throws:** [IllegalMonitorStateException](http://docs.google.com/java/lang/IllegalMonitorStateException.html) - if acquiring would place this synchronizer in an illegal state. This exception must be thrown in a consistent fashion for synchronization to work correctly. [UnsupportedOperationException](http://docs.google.com/java/lang/UnsupportedOperationException.html) - if exclusive mode is not supported

### tryRelease

protected boolean **tryRelease**(int arg)

Attempts to set the state to reflect a release in exclusive mode.

This method is always invoked by the thread performing release.

The default implementation throws [UnsupportedOperationException](http://docs.google.com/java/lang/UnsupportedOperationException.html).

**Parameters:**arg - the release argument. This value is always the one passed to a release method, or the current state value upon entry to a condition wait. The value is otherwise uninterpreted and can represent anything you like. **Returns:**true if this object is now in a fully released state, so that any waiting threads may attempt to acquire; and false otherwise. **Throws:** [IllegalMonitorStateException](http://docs.google.com/java/lang/IllegalMonitorStateException.html) - if releasing would place this synchronizer in an illegal state. This exception must be thrown in a consistent fashion for synchronization to work correctly. [UnsupportedOperationException](http://docs.google.com/java/lang/UnsupportedOperationException.html) - if exclusive mode is not supported

### tryAcquireShared

protected int **tryAcquireShared**(int arg)

Attempts to acquire in shared mode. This method should query if the state of the object permits it to be acquired in the shared mode, and if so to acquire it.

This method is always invoked by the thread performing acquire. If this method reports failure, the acquire method may queue the thread, if it is not already queued, until it is signalled by a release from some other thread.

The default implementation throws [UnsupportedOperationException](http://docs.google.com/java/lang/UnsupportedOperationException.html).

**Parameters:**arg - the acquire argument. This value is always the one passed to an acquire method, or is the value saved on entry to a condition wait. The value is otherwise uninterpreted and can represent anything you like. **Returns:**a negative value on failure; zero if acquisition in shared mode succeeded but no subsequent shared-mode acquire can succeed; and a positive value if acquisition in shared mode succeeded and subsequent shared-mode acquires might also succeed, in which case a subsequent waiting thread must check availability. (Support for three different return values enables this method to be used in contexts where acquires only sometimes act exclusively.) Upon success, this object has been acquired. **Throws:** [IllegalMonitorStateException](http://docs.google.com/java/lang/IllegalMonitorStateException.html) - if acquiring would place this synchronizer in an illegal state. This exception must be thrown in a consistent fashion for synchronization to work correctly. [UnsupportedOperationException](http://docs.google.com/java/lang/UnsupportedOperationException.html) - if shared mode is not supported

### tryReleaseShared

protected boolean **tryReleaseShared**(int arg)

Attempts to set the state to reflect a release in shared mode.

This method is always invoked by the thread performing release.

The default implementation throws [UnsupportedOperationException](http://docs.google.com/java/lang/UnsupportedOperationException.html).

**Parameters:**arg - the release argument. This value is always the one passed to a release method, or the current state value upon entry to a condition wait. The value is otherwise uninterpreted and can represent anything you like. **Returns:**true if this release of shared mode may permit a waiting acquire (shared or exclusive) to succeed; and false otherwise **Throws:** [IllegalMonitorStateException](http://docs.google.com/java/lang/IllegalMonitorStateException.html) - if releasing would place this synchronizer in an illegal state. This exception must be thrown in a consistent fashion for synchronization to work correctly. [UnsupportedOperationException](http://docs.google.com/java/lang/UnsupportedOperationException.html) - if shared mode is not supported

### isHeldExclusively

protected boolean **isHeldExclusively**()

Returns true if synchronization is held exclusively with respect to the current (calling) thread. This method is invoked upon each call to a non-waiting [AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) method. (Waiting methods instead invoke [release(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#release(int)).)

The default implementation throws [UnsupportedOperationException](http://docs.google.com/java/lang/UnsupportedOperationException.html). This method is invoked internally only within [AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) methods, so need not be defined if conditions are not used.

**Returns:**true if synchronization is held exclusively; false otherwise **Throws:** [UnsupportedOperationException](http://docs.google.com/java/lang/UnsupportedOperationException.html) - if conditions are not supported

### acquire

public final void **acquire**(int arg)

Acquires in exclusive mode, ignoring interrupts. Implemented by invoking at least once [tryAcquire(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquire(int)), returning on success. Otherwise the thread is queued, possibly repeatedly blocking and unblocking, invoking [tryAcquire(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquire(int)) until success. This method can be used to implement method [Lock.lock()](http://docs.google.com/java/util/concurrent/locks/Lock.html#lock()).

**Parameters:**arg - the acquire argument. This value is conveyed to [tryAcquire(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquire(int)) but is otherwise uninterpreted and can represent anything you like.

### acquireInterruptibly

public final void **acquireInterruptibly**(int arg)  
 throws [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html)

Acquires in exclusive mode, aborting if interrupted. Implemented by first checking interrupt status, then invoking at least once [tryAcquire(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquire(int)), returning on success. Otherwise the thread is queued, possibly repeatedly blocking and unblocking, invoking [tryAcquire(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquire(int)) until success or the thread is interrupted. This method can be used to implement method [Lock.lockInterruptibly()](http://docs.google.com/java/util/concurrent/locks/Lock.html#lockInterruptibly()).

**Parameters:**arg - the acquire argument. This value is conveyed to [tryAcquire(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquire(int)) but is otherwise uninterpreted and can represent anything you like. **Throws:** [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) - if the current thread is interrupted

### tryAcquireNanos

public final boolean **tryAcquireNanos**(int arg,  
 long nanosTimeout)  
 throws [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html)

Attempts to acquire in exclusive mode, aborting if interrupted, and failing if the given timeout elapses. Implemented by first checking interrupt status, then invoking at least once [tryAcquire(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquire(int)), returning on success. Otherwise, the thread is queued, possibly repeatedly blocking and unblocking, invoking [tryAcquire(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquire(int)) until success or the thread is interrupted or the timeout elapses. This method can be used to implement method [Lock.tryLock(long, TimeUnit)](http://docs.google.com/java/util/concurrent/locks/Lock.html#tryLock(long,%20java.util.concurrent.TimeUnit)).

**Parameters:**arg - the acquire argument. This value is conveyed to [tryAcquire(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquire(int)) but is otherwise uninterpreted and can represent anything you like.nanosTimeout - the maximum number of nanoseconds to wait **Returns:**true if acquired; false if timed out **Throws:** [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) - if the current thread is interrupted

### release

public final boolean **release**(int arg)

Releases in exclusive mode. Implemented by unblocking one or more threads if [tryRelease(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryRelease(int)) returns true. This method can be used to implement method [Lock.unlock()](http://docs.google.com/java/util/concurrent/locks/Lock.html#unlock()).

**Parameters:**arg - the release argument. This value is conveyed to [tryRelease(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryRelease(int)) but is otherwise uninterpreted and can represent anything you like. **Returns:**the value returned from [tryRelease(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryRelease(int))

### acquireShared

public final void **acquireShared**(int arg)

Acquires in shared mode, ignoring interrupts. Implemented by first invoking at least once [tryAcquireShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireShared(int)), returning on success. Otherwise the thread is queued, possibly repeatedly blocking and unblocking, invoking [tryAcquireShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireShared(int)) until success.

**Parameters:**arg - the acquire argument. This value is conveyed to [tryAcquireShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireShared(int)) but is otherwise uninterpreted and can represent anything you like.

### acquireSharedInterruptibly

public final void **acquireSharedInterruptibly**(int arg)  
 throws [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html)

Acquires in shared mode, aborting if interrupted. Implemented by first checking interrupt status, then invoking at least once [tryAcquireShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireShared(int)), returning on success. Otherwise the thread is queued, possibly repeatedly blocking and unblocking, invoking [tryAcquireShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireShared(int)) until success or the thread is interrupted.

**Parameters:**arg - the acquire argument. This value is conveyed to [tryAcquireShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireShared(int)) but is otherwise uninterpreted and can represent anything you like. **Throws:** [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) - if the current thread is interrupted

### tryAcquireSharedNanos

public final boolean **tryAcquireSharedNanos**(int arg,  
 long nanosTimeout)  
 throws [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html)

Attempts to acquire in shared mode, aborting if interrupted, and failing if the given timeout elapses. Implemented by first checking interrupt status, then invoking at least once [tryAcquireShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireShared(int)), returning on success. Otherwise, the thread is queued, possibly repeatedly blocking and unblocking, invoking [tryAcquireShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireShared(int)) until success or the thread is interrupted or the timeout elapses.

**Parameters:**arg - the acquire argument. This value is conveyed to [tryAcquireShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryAcquireShared(int)) but is otherwise uninterpreted and can represent anything you like.nanosTimeout - the maximum number of nanoseconds to wait **Returns:**true if acquired; false if timed out **Throws:** [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) - if the current thread is interrupted

### releaseShared

public final boolean **releaseShared**(int arg)

Releases in shared mode. Implemented by unblocking one or more threads if [tryReleaseShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryReleaseShared(int)) returns true.

**Parameters:**arg - the release argument. This value is conveyed to [tryReleaseShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryReleaseShared(int)) but is otherwise uninterpreted and can represent anything you like. **Returns:**the value returned from [tryReleaseShared(int)](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#tryReleaseShared(int))

### hasQueuedThreads

public final boolean **hasQueuedThreads**()

Queries whether any threads are waiting to acquire. Note that because cancellations due to interrupts and timeouts may occur at any time, a true return does not guarantee that any other thread will ever acquire.

In this implementation, this operation returns in constant time.

**Returns:**true if there may be other threads waiting to acquire

### hasContended

public final boolean **hasContended**()

Queries whether any threads have ever contended to acquire this synchronizer; that is if an acquire method has ever blocked.

In this implementation, this operation returns in constant time.

**Returns:**true if there has ever been contention

### getFirstQueuedThread

public final [Thread](http://docs.google.com/java/lang/Thread.html) **getFirstQueuedThread**()

Returns the first (longest-waiting) thread in the queue, or null if no threads are currently queued.

In this implementation, this operation normally returns in constant time, but may iterate upon contention if other threads are concurrently modifying the queue.

**Returns:**the first (longest-waiting) thread in the queue, or null if no threads are currently queued

### isQueued

public final boolean **isQueued**([Thread](http://docs.google.com/java/lang/Thread.html) thread)

Returns true if the given thread is currently queued.

This implementation traverses the queue to determine presence of the given thread.

**Parameters:**thread - the thread **Returns:**true if the given thread is on the queue **Throws:** [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if the thread is null

### getQueueLength

public final int **getQueueLength**()

Returns an estimate of the number of threads waiting to acquire. The value is only an estimate because the number of threads may change dynamically while this method traverses internal data structures. This method is designed for use in monitoring system state, not for synchronization control.

**Returns:**the estimated number of threads waiting to acquire

### getQueuedThreads

public final [Collection](http://docs.google.com/java/util/Collection.html)<[Thread](http://docs.google.com/java/lang/Thread.html)> **getQueuedThreads**()

Returns a collection containing threads that may be waiting to acquire. Because the actual set of threads may change dynamically while constructing this result, the returned collection is only a best-effort estimate. The elements of the returned collection are in no particular order. This method is designed to facilitate construction of subclasses that provide more extensive monitoring facilities.

**Returns:**the collection of threads

### getExclusiveQueuedThreads

public final [Collection](http://docs.google.com/java/util/Collection.html)<[Thread](http://docs.google.com/java/lang/Thread.html)> **getExclusiveQueuedThreads**()

Returns a collection containing threads that may be waiting to acquire in exclusive mode. This has the same properties as [getQueuedThreads()](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getQueuedThreads()) except that it only returns those threads waiting due to an exclusive acquire.

**Returns:**the collection of threads

### getSharedQueuedThreads

public final [Collection](http://docs.google.com/java/util/Collection.html)<[Thread](http://docs.google.com/java/lang/Thread.html)> **getSharedQueuedThreads**()

Returns a collection containing threads that may be waiting to acquire in shared mode. This has the same properties as [getQueuedThreads()](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getQueuedThreads()) except that it only returns those threads waiting due to a shared acquire.

**Returns:**the collection of threads

### toString

public [String](http://docs.google.com/java/lang/String.html) **toString**()

Returns a string identifying this synchronizer, as well as its state. The state, in brackets, includes the String "State =" followed by the current value of [getState()](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.html#getState()), and either "nonempty" or "empty" depending on whether the queue is empty.

**Overrides:**[toString](http://docs.google.com/java/lang/Object.html#toString()) in class [Object](http://docs.google.com/java/lang/Object.html) **Returns:**a string identifying this synchronizer, as well as its state

### owns

public final boolean **owns**([AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) condition)

Queries whether the given ConditionObject uses this synchronizer as its lock.

**Parameters:**condition - the condition **Returns:**true if owned **Throws:** [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if the condition is null

### hasWaiters

public final boolean **hasWaiters**([AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) condition)

Queries whether any threads are waiting on the given condition associated with this synchronizer. Note that because timeouts and interrupts may occur at any time, a true return does not guarantee that a future signal will awaken any threads. This method is designed primarily for use in monitoring of the system state.

**Parameters:**condition - the condition **Returns:**true if there are any waiting threads **Throws:** [IllegalMonitorStateException](http://docs.google.com/java/lang/IllegalMonitorStateException.html) - if exclusive synchronization is not held [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if the given condition is not associated with this synchronizer [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if the condition is null

### getWaitQueueLength

public final int **getWaitQueueLength**([AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) condition)

Returns an estimate of the number of threads waiting on the given condition associated with this synchronizer. Note that because timeouts and interrupts may occur at any time, the estimate serves only as an upper bound on the actual number of waiters. This method is designed for use in monitoring of the system state, not for synchronization control.

**Parameters:**condition - the condition **Returns:**the estimated number of waiting threads **Throws:** [IllegalMonitorStateException](http://docs.google.com/java/lang/IllegalMonitorStateException.html) - if exclusive synchronization is not held [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if the given condition is not associated with this synchronizer [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if the condition is null

### getWaitingThreads

public final [Collection](http://docs.google.com/java/util/Collection.html)<[Thread](http://docs.google.com/java/lang/Thread.html)> **getWaitingThreads**([AbstractQueuedSynchronizer.ConditionObject](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) condition)

Returns a collection containing those threads that may be waiting on the given condition associated with this synchronizer. Because the actual set of threads may change dynamically while constructing this result, the returned collection is only a best-effort estimate. The elements of the returned collection are in no particular order.

**Parameters:**condition - the condition **Returns:**the collection of threads **Throws:** [IllegalMonitorStateException](http://docs.google.com/java/lang/IllegalMonitorStateException.html) - if exclusive synchronization is not held [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if the given condition is not associated with this synchronizer [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if the condition is null

| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/AbstractQueuedSynchronizer.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/AbstractQueuedLongSynchronizer.ConditionObject.html)   [**NEXT CLASS**](http://docs.google.com/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/locks/AbstractQueuedSynchronizer.html)    [**NO FRAMES**](http://docs.google.com/AbstractQueuedSynchronizer.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |
| SUMMARY: [NESTED](#3znysh7) | FIELD | [CONSTR](#2et92p0) | [METHOD](#tyjcwt) | DETAIL: FIELD | [CONSTR](#4d34og8) | [METHOD](#17dp8vu) |

[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).