| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/Lock.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/Condition.html)   [**NEXT CLASS**](http://docs.google.com/java/util/concurrent/locks/LockSupport.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/locks/Lock.html)    [**NO FRAMES**](http://docs.google.com/Lock.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 Lock

**All Known Implementing Classes:** [ReentrantLock](http://docs.google.com/java/util/concurrent/locks/ReentrantLock.html), [ReentrantReadWriteLock.ReadLock](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.ReadLock.html), [ReentrantReadWriteLock.WriteLock](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.WriteLock.html)

public interface **Lock**

Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements. They allow more flexible structuring, may have quite different properties, and may support multiple associated [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) objects.

A lock is a tool for controlling access to a shared resource by multiple threads. Commonly, a lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and all access to the shared resource requires that the lock be acquired first. However, some locks may allow concurrent access to a shared resource, such as the read lock of a [ReadWriteLock](http://docs.google.com/java/util/concurrent/locks/ReadWriteLock.html).

The use of synchronized methods or statements provides access to the implicit monitor lock associated with every object, but forces all lock acquisition and release to occur in a block-structured way: when multiple locks are acquired they must be released in the opposite order, and all locks must be released in the same lexical scope in which they were acquired.

While the scoping mechanism for synchronized methods and statements makes it much easier to program with monitor locks, and helps avoid many common programming errors involving locks, there are occasions where you need to work with locks in a more flexible way. For example, some algorithms for traversing concurrently accessed data structures require the use of "hand-over-hand" or "chain locking": you acquire the lock of node A, then node B, then release A and acquire C, then release B and acquire D and so on. Implementations of the Lock interface enable the use of such techniques by allowing a lock to be acquired and released in different scopes, and allowing multiple locks to be acquired and released in any order.

With this increased flexibility comes additional responsibility. The absence of block-structured locking removes the automatic release of locks that occurs with synchronized methods and statements. In most cases, the following idiom should be used:

Lock l = ...;  
 l.lock();  
 try {  
 // access the resource protected by this lock  
 } finally {  
 l.unlock();  
 }

When locking and unlocking occur in different scopes, care must be taken to ensure that all code that is executed while the lock is held is protected by try-finally or try-catch to ensure that the lock is released when necessary.

Lock implementations provide additional functionality over the use of synchronized methods and statements by providing a non-blocking attempt to acquire a lock ([tryLock()](http://docs.google.com/java/util/concurrent/locks/Lock.html#tryLock())), an attempt to acquire the lock that can be interrupted ([lockInterruptibly()](http://docs.google.com/java/util/concurrent/locks/Lock.html#lockInterruptibly()), and an attempt to acquire the lock that can timeout ([tryLock(long, TimeUnit)](http://docs.google.com/java/util/concurrent/locks/Lock.html#tryLock(long,%20java.util.concurrent.TimeUnit))).

A Lock class can also provide behavior and semantics that is quite different from that of the implicit monitor lock, such as guaranteed ordering, non-reentrant usage, or deadlock detection. If an implementation provides such specialized semantics then the implementation must document those semantics.

Note that Lock instances are just normal objects and can themselves be used as the target in a synchronized statement. Acquiring the monitor lock of a Lock instance has no specified relationship with invoking any of the [lock()](http://docs.google.com/java/util/concurrent/locks/Lock.html#lock()) methods of that instance. It is recommended that to avoid confusion you never use Lock instances in this way, except 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.

### Memory Synchronization

All Lock implementations *must* enforce the same memory synchronization semantics as provided by the built-in monitor lock, as described in  [The Java Language Specification, Third Edition (17.4 Memory Model)](http://java.sun.com/docs/books/jls/):

* A successful lock operation has the same memory synchronization effects as a successful *Lock* action.
* A successful unlock operation has the same memory synchronization effects as a successful *Unlock* action.

Unsuccessful locking and unlocking operations, and reentrant locking/unlocking operations, do not require any memory synchronization effects.

### Implementation Considerations

The three forms of lock acquisition (interruptible, non-interruptible, and timed) may differ in their performance characteristics, ordering guarantees, or other implementation qualities. Further, the ability to interrupt the *ongoing* acquisition of a lock may not be available in a given Lock class. Consequently, an implementation is not required to define exactly the same guarantees or semantics for all three forms of lock acquisition, nor is it required to support interruption of an ongoing lock acquisition. An implementation is required to clearly document the semantics and guarantees provided by each of the locking methods. It must also obey the interruption semantics as defined in this interface, to the extent that interruption of lock acquisition is supported: which is either totally, or only on method entry.

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 **See Also:**[ReentrantLock](http://docs.google.com/java/util/concurrent/locks/ReentrantLock.html), [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html), [ReadWriteLock](http://docs.google.com/java/util/concurrent/locks/ReadWriteLock.html)

| **Method Summary** | |
| --- | --- |
| void | [**lock**](http://docs.google.com/java/util/concurrent/locks/Lock.html#lock())()            Acquires the lock. |
| void | [**lockInterruptibly**](http://docs.google.com/java/util/concurrent/locks/Lock.html#lockInterruptibly())()            Acquires the lock unless the current thread is [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()). |
| [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) | [**newCondition**](http://docs.google.com/java/util/concurrent/locks/Lock.html#newCondition())()            Returns a new [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) instance that is bound to this Lock instance. |
| boolean | [**tryLock**](http://docs.google.com/java/util/concurrent/locks/Lock.html#tryLock())()            Acquires the lock only if it is free at the time of invocation. |
| boolean | [**tryLock**](http://docs.google.com/java/util/concurrent/locks/Lock.html#tryLock(long,%20java.util.concurrent.TimeUnit))(long time, [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit)            Acquires the lock if it is free within the given waiting time and the current thread has not been [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()). |
| void | [**unlock**](http://docs.google.com/java/util/concurrent/locks/Lock.html#unlock())()            Releases the lock. |

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

### lock

void **lock**()

Acquires the lock.

If the lock is not available then the current thread becomes disabled for thread scheduling purposes and lies dormant until the lock has been acquired.

**Implementation Considerations**

A Lock implementation may be able to detect erroneous use of the lock, such as an invocation that would cause deadlock, and may throw an (unchecked) exception in such circumstances. The circumstances and the exception type must be documented by that Lock implementation.

### lockInterruptibly

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

Acquires the lock unless the current thread is [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()).

Acquires the lock if it is available and returns immediately.

If the lock is not available then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of two things happens:

* The lock is acquired by the current thread; or
* Some other thread [interrupts](http://docs.google.com/java/lang/Thread.html#interrupt()) the current thread, and interruption of lock acquisition is supported.

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 acquiring the lock, and interruption of lock acquisition is supported,

then [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) is thrown and the current thread's interrupted status is cleared.

**Implementation Considerations**

The ability to interrupt a lock acquisition in some implementations may not be possible, and if possible may be an expensive operation. The programmer should be aware that this may be the case. An implementation should document when this is the case.

An implementation can favor responding to an interrupt over normal method return.

A Lock implementation may be able to detect erroneous use of the lock, such as an invocation that would cause deadlock, and may throw an (unchecked) exception in such circumstances. The circumstances and the exception type must be documented by that Lock implementation.

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

### tryLock

boolean **tryLock**()

Acquires the lock only if it is free at the time of invocation.

Acquires the lock if it is available and returns immediately with the value true. If the lock is not available then this method will return immediately with the value false.

A typical usage idiom for this method would be:

Lock lock = ...;  
 if (lock.tryLock()) {  
 try {  
 // manipulate protected state  
 } finally {  
 lock.unlock();  
 }  
 } else {  
 // perform alternative actions  
 }

This usage ensures that the lock is unlocked if it was acquired, and doesn't try to unlock if the lock was not acquired.

**Returns:**true if the lock was acquired and false otherwise

### tryLock

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

Acquires the lock if it is free within the given waiting time and the current thread has not been [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()).

If the lock is available this method returns immediately with the value true. If the lock is not available then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of three things happens:

* The lock is acquired by the current thread; or
* Some other thread [interrupts](http://docs.google.com/java/lang/Thread.html#interrupt()) the current thread, and interruption of lock acquisition is supported; or
* The specified waiting time elapses

If the lock is acquired then the value true is returned.

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 acquiring the lock, and interruption of lock acquisition is supported,

then [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) is thrown and the current thread's interrupted status is cleared.

If the specified waiting time elapses then the value false is returned. If the time is less than or equal to zero, the method will not wait at all.

**Implementation Considerations**

The ability to interrupt a lock acquisition in some implementations may not be possible, and if possible may be an expensive operation. The programmer should be aware that this may be the case. An implementation should document when this is the case.

An implementation can favor responding to an interrupt over normal method return, or reporting a timeout.

A Lock implementation may be able to detect erroneous use of the lock, such as an invocation that would cause deadlock, and may throw an (unchecked) exception in such circumstances. The circumstances and the exception type must be documented by that Lock implementation.

**Parameters:**time - the maximum time to wait for the lockunit - the time unit of the time argument **Returns:**true if the lock was acquired and false if the waiting time elapsed before the lock was acquired **Throws:** [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) - if the current thread is interrupted while acquiring the lock (and interruption of lock acquisition is supported)

### unlock

void **unlock**()

Releases the lock.

**Implementation Considerations**

A Lock implementation will usually impose restrictions on which thread can release a lock (typically only the holder of the lock can release it) and may throw an (unchecked) exception if the restriction is violated. Any restrictions and the exception type must be documented by that Lock implementation.

### newCondition

[Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) **newCondition**()

Returns a new [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) instance that is bound to this Lock instance.

Before waiting on the condition the lock must be held by the current thread. A call to [Condition.await()](http://docs.google.com/java/util/concurrent/locks/Condition.html#await()) will atomically release the lock before waiting and re-acquire the lock before the wait returns.

**Implementation Considerations**

The exact operation of the [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) instance depends on the Lock implementation and must be documented by that implementation.

**Returns:**A new [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) instance for this Lock instance **Throws:** [UnsupportedOperationException](http://docs.google.com/java/lang/UnsupportedOperationException.html) - if this Lock implementation does not support conditions

| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/Lock.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/Condition.html)   [**NEXT CLASS**](http://docs.google.com/java/util/concurrent/locks/LockSupport.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/locks/Lock.html)    [**NO FRAMES**](http://docs.google.com/Lock.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).