<http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html>

java.util.concurrent.locks

**Interface Lock**

* **All Known Implementing Classes:**

[ReentrantLock](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html), [ReentrantReadWriteLock.ReadLock](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.ReadLock.html), [ReentrantReadWriteLock.WriteLock](http://docs.oracle.com/javase/7/docs/api/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.oracle.com/javase/7/docs/api/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.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html" \o "interface in java.util.concurrent.locks).

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.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html" \l "tryLock())), an attempt to acquire the lock that can be interrupted ([lockInterruptibly()](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html" \l "lockInterruptibly()), and an attempt to acquire the lock that can timeout ([tryLock(long, TimeUnit)](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html" \l "tryLock(long, java.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.oracle.com/javase/7/docs/api/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.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html" \o "class in java.lang) 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 section 17.4 of *The Java™ Language Specification*:

* + 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.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html), [Condition](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html), [ReadWriteLock](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html" \o "interface in java.util.concurrent.locks)

### Method Summary

|  |  |
| --- | --- |
| **Methods** | |
| **Modifier and Type** | **Method and Description** |
| void | [**lock**](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html#lock())()  Acquires the lock. |
| void | [**lockInterruptibly**](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html#lockInterruptibly())()  Acquires the lock unless the current thread is [**interrupted**](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()). |
| [**Condition**](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html) | [**newCondition**](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html#newCondition())()  Returns a new [**Condition**](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html) instance that is bound to this Lock instance. |
| boolean | [**tryLock**](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html#tryLock())()  Acquires the lock only if it is free at the time of invocation. |
| boolean | [**tryLock**](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html#tryLock(long, java.util.concurrent.TimeUnit))(long time, **[TimeUnit](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeUnit.html" \o "enum in java.util.concurrent)** unit)  Acquires the lock if it is free within the given waiting time and the current thread has not been [**interrupted**](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()). |
| void | [**unlock**](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html#unlock())()  Releases the lock. |

<http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html>

java.util.concurrent.locks

**Class ReentrantLock**

* [java.lang.Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)
  + java.util.concurrent.locks.ReentrantLock
* **All Implemented Interfaces:**

[Serializable](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html), [Lock](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html)

public class **ReentrantLock**

extends [Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)

implements [Lock](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html), [Serializable](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

A reentrant mutual exclusion [Lock](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html) with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.

A ReentrantLock is *owned* by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock. This can be checked using methods [isHeldByCurrentThread()](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html" \l "isHeldByCurrentThread()), and [getHoldCount()](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html" \l "getHoldCount()).

The constructor for this class accepts an optional *fairness* parameter. When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order. Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation. Note however, that fairness of locks does not guarantee fairness of thread scheduling. Thus, one of many threads using a fair lock may obtain it multiple times in succession while other active threads are not progressing and not currently holding the lock. Also note that the untimed [tryLock](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html" \l "tryLock()) method does not honor the fairness setting. It will succeed if the lock is available even if other threads are waiting.

It is recommended practice to *always* immediately follow a call to lock with a try block, most typically in a before/after construction such as:

class X {

private final ReentrantLock lock = new ReentrantLock();

// ...

public void m() {

lock.lock(); // block until condition holds

try {

// ... method body

} finally {

lock.unlock();

}

}

}

In addition to implementing the [Lock](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html) interface, this class defines methods isLocked and getLockQueueLength, as well as some associated protected access methods that may be useful for instrumentation and monitoring.

Serialization of this class behaves in the same way as built-in locks: a deserialized lock is in the unlocked state, regardless of its state when serialized.

This lock supports a maximum of 2147483647 recursive locks by the same thread. Attempts to exceed this limit result in [Error](http://docs.oracle.com/javase/7/docs/api/java/lang/Error.html) throws from locking methods.

**Since:**

1.5

**See Also:**

[Serialized Form](http://docs.oracle.com/javase/7/docs/api/serialized-form.html#java.util.concurrent.locks.ReentrantLock)

* + **Constructor Summary**