| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/ReentrantReadWriteLock.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/ReentrantLock.html)   [**NEXT CLASS**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.ReadLock.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/locks/ReentrantReadWriteLock.html)    [**NO FRAMES**](http://docs.google.com/ReentrantReadWriteLock.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |
| SUMMARY: [NESTED](#3znysh7) | FIELD | [CONSTR](#2et92p0) | [METHOD](#tyjcwt) | DETAIL: FIELD | [CONSTR](#1t3h5sf) | [METHOD](#17dp8vu) |

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

Class ReentrantReadWriteLock

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

**All Implemented Interfaces:** [Serializable](http://docs.google.com/java/io/Serializable.html), [ReadWriteLock](http://docs.google.com/java/util/concurrent/locks/ReadWriteLock.html)

public class **ReentrantReadWriteLock**extends [Object](http://docs.google.com/java/lang/Object.html)implements [ReadWriteLock](http://docs.google.com/java/util/concurrent/locks/ReadWriteLock.html), [Serializable](http://docs.google.com/java/io/Serializable.html)

An implementation of [ReadWriteLock](http://docs.google.com/java/util/concurrent/locks/ReadWriteLock.html) supporting similar semantics to [ReentrantLock](http://docs.google.com/java/util/concurrent/locks/ReentrantLock.html).

This class has the following properties:

* **Acquisition order**  
  This class does not impose a reader or writer preference ordering for lock access. However, it does support an optional *fairness* policy.***Non-fair mode (default)*** When constructed as non-fair (the default), the order of entry to the read and write lock is unspecified, subject to reentrancy constraints. A nonfair lock that is continously contended may indefinitely postpone one or more reader or writer threads, but will normally have higher throughput than a fair lock.  
  ***Fair mode*** When constructed as fair, threads contend for entry using an approximately arrival-order policy. When the currently held lock is released either the longest-waiting single writer thread will be assigned the write lock, or if there is a group of reader threads waiting longer than all waiting writer threads, that group will be assigned the read lock.  
  A thread that tries to acquire a fair read lock (non-reentrantly) will block if either the write lock is held, or there is a waiting writer thread. The thread will not acquire the read lock until after the oldest currently waiting writer thread has acquired and released the write lock. Of course, if a waiting writer abandons its wait, leaving one or more reader threads as the longest waiters in the queue with the write lock free, then those readers will be assigned the read lock.  
  A thread that tries to acquire a fair write lock (non-reentrantly) will block unless both the read lock and write lock are free (which implies there are no waiting threads). (Note that the non-blocking [ReentrantReadWriteLock.ReadLock.tryLock()](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.ReadLock.html#tryLock()) and [ReentrantReadWriteLock.WriteLock.tryLock()](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.WriteLock.html#tryLock()) methods do not honor this fair setting and will acquire the lock if it is possible, regardless of waiting threads.)
* **Reentrancy**  
  This lock allows both readers and writers to reacquire read or write locks in the style of a [ReentrantLock](http://docs.google.com/java/util/concurrent/locks/ReentrantLock.html). Non-reentrant readers are not allowed until all write locks held by the writing thread have been released.  
  Additionally, a writer can acquire the read lock, but not vice-versa. Among other applications, reentrancy can be useful when write locks are held during calls or callbacks to methods that perform reads under read locks. If a reader tries to acquire the write lock it will never succeed.
* **Lock downgrading**  
  Reentrancy also allows downgrading from the write lock to a read lock, by acquiring the write lock, then the read lock and then releasing the write lock. However, upgrading from a read lock to the write lock is **not** possible.
* **Interruption of lock acquisition**  
  The read lock and write lock both support interruption during lock acquisition.
* [**Condition**](http://docs.google.com/java/util/concurrent/locks/Condition.html) **support**  
  The write lock provides a [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) implementation that behaves in the same way, with respect to the write lock, as the [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) implementation provided by [ReentrantLock.newCondition()](http://docs.google.com/java/util/concurrent/locks/ReentrantLock.html#newCondition()) does for [ReentrantLock](http://docs.google.com/java/util/concurrent/locks/ReentrantLock.html). This [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) can, of course, only be used with the write lock.  
  The read lock does not support a [Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) and readLock().newCondition() throws UnsupportedOperationException.
* **Instrumentation**  
  This class supports methods to determine whether locks are held or contended. These methods are designed for monitoring system state, not for synchronization control.

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.

**Sample usages**. Here is a code sketch showing how to exploit reentrancy to perform lock downgrading after updating a cache (exception handling is elided for simplicity):

class CachedData {  
 Object data;  
 volatile boolean cacheValid;  
 ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();  
  
 void processCachedData() {  
 rwl.readLock().lock();  
 if (!cacheValid) {  
 // Must release read lock before acquiring write lock  
 rwl.readLock().unlock();  
 rwl.writeLock().lock();  
 // Recheck state because another thread might have acquired  
 // write lock and changed state before we did.  
 if (!cacheValid) {  
 data = ...  
 cacheValid = true;  
 }  
 // Downgrade by acquiring read lock before releasing write lock  
 rwl.readLock().lock();  
 rwl.writeLock().unlock(); // Unlock write, still hold read  
 }  
  
 use(data);  
 rwl.readLock().unlock();  
 }  
 }

ReentrantReadWriteLocks can be used to improve concurrency in some uses of some kinds of Collections. This is typically worthwhile only when the collections are expected to be large, accessed by more reader threads than writer threads, and entail operations with overhead that outweighs synchronization overhead. For example, here is a class using a TreeMap that is expected to be large and concurrently accessed.

class RWDictionary {  
 private final Map<String, Data> m = new TreeMap<String, Data>();  
 private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();  
 private final Lock r = rwl.readLock();  
 private final Lock w = rwl.writeLock();  
  
 public Data get(String key) {  
 r.lock();  
 try { return m.get(key); }  
 finally { r.unlock(); }  
 }  
 public String[] allKeys() {  
 r.lock();  
 try { return m.keySet().toArray(); }  
 finally { r.unlock(); }  
 }  
 public Data put(String key, Data value) {  
 w.lock();  
 try { return m.put(key, value); }  
 finally { w.unlock(); }  
 }  
 public void clear() {  
 w.lock();  
 try { m.clear(); }  
 finally { w.unlock(); }  
 }  
 }

### Implementation Notes

This lock supports a maximum of 65535 recursive write locks and 65535 read locks. Attempts to exceed these limits result in [Error](http://docs.google.com/java/lang/Error.html) throws from locking methods.

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

| **Nested Class Summary** | |
| --- | --- |
| static class | [**ReentrantReadWriteLock.ReadLock**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.ReadLock.html)            The lock returned by method [readLock()](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#readLock()). |
| static class | [**ReentrantReadWriteLock.WriteLock**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.WriteLock.html)            The lock returned by method [writeLock()](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#writeLock()). |

| **Constructor Summary** | |
| --- | --- |
| [**ReentrantReadWriteLock**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#ReentrantReadWriteLock())()            Creates a new ReentrantReadWriteLock with default (nonfair) ordering properties. |
| [**ReentrantReadWriteLock**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#ReentrantReadWriteLock(boolean))(boolean fair)            Creates a new ReentrantReadWriteLock with the given fairness policy. |

| **Method Summary** | |
| --- | --- |
| protected  [Thread](http://docs.google.com/java/lang/Thread.html) | [**getOwner**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#getOwner())()            Returns the thread that currently owns the write lock, or null if not owned. |
| protected  [Collection](http://docs.google.com/java/util/Collection.html)<[Thread](http://docs.google.com/java/lang/Thread.html)> | [**getQueuedReaderThreads**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#getQueuedReaderThreads())()            Returns a collection containing threads that may be waiting to acquire the read lock. |
| protected  [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/ReentrantReadWriteLock.html#getQueuedThreads())()            Returns a collection containing threads that may be waiting to acquire either the read or write lock. |
| protected  [Collection](http://docs.google.com/java/util/Collection.html)<[Thread](http://docs.google.com/java/lang/Thread.html)> | [**getQueuedWriterThreads**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#getQueuedWriterThreads())()            Returns a collection containing threads that may be waiting to acquire the write lock. |
| int | [**getQueueLength**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#getQueueLength())()            Returns an estimate of the number of threads waiting to acquire either the read or write lock. |
| int | [**getReadHoldCount**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#getReadHoldCount())()            Queries the number of reentrant read holds on this lock by the current thread. |
| int | [**getReadLockCount**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#getReadLockCount())()            Queries the number of read locks held for this lock. |
| protected  [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/ReentrantReadWriteLock.html#getWaitingThreads(java.util.concurrent.locks.Condition))([Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) condition)            Returns a collection containing those threads that may be waiting on the given condition associated with the write lock. |
| int | [**getWaitQueueLength**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#getWaitQueueLength(java.util.concurrent.locks.Condition))([Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) condition)            Returns an estimate of the number of threads waiting on the given condition associated with the write lock. |
| int | [**getWriteHoldCount**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#getWriteHoldCount())()            Queries the number of reentrant write holds on this lock by the current thread. |
| boolean | [**hasQueuedThread**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#hasQueuedThread(java.lang.Thread))([Thread](http://docs.google.com/java/lang/Thread.html) thread)            Queries whether the given thread is waiting to acquire either the read or write lock. |
| boolean | [**hasQueuedThreads**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#hasQueuedThreads())()            Queries whether any threads are waiting to acquire the read or write lock. |
| boolean | [**hasWaiters**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#hasWaiters(java.util.concurrent.locks.Condition))([Condition](http://docs.google.com/java/util/concurrent/locks/Condition.html) condition)            Queries whether any threads are waiting on the given condition associated with the write lock. |
| boolean | [**isFair**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#isFair())()            Returns true if this lock has fairness set true. |
| boolean | [**isWriteLocked**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#isWriteLocked())()            Queries if the write lock is held by any thread. |
| boolean | [**isWriteLockedByCurrentThread**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#isWriteLockedByCurrentThread())()            Queries if the write lock is held by the current thread. |
| [ReentrantReadWriteLock.ReadLock](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.ReadLock.html) | [**readLock**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#readLock())()            Returns the lock used for reading. |
| [String](http://docs.google.com/java/lang/String.html) | [**toString**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#toString())()            Returns a string identifying this lock, as well as its lock state. |
| [ReentrantReadWriteLock.WriteLock](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.WriteLock.html) | [**writeLock**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.html#writeLock())()            Returns the lock used for writing. |

| **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** |
| --- |

### ReentrantReadWriteLock

public **ReentrantReadWriteLock**()

Creates a new ReentrantReadWriteLock with default (nonfair) ordering properties.

### ReentrantReadWriteLock

public **ReentrantReadWriteLock**(boolean fair)

Creates a new ReentrantReadWriteLock with the given fairness policy.

**Parameters:**fair - true if this lock should use a fair ordering policy

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

### writeLock

public [ReentrantReadWriteLock.WriteLock](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.WriteLock.html) **writeLock**()

**Description copied from interface:** [**ReadWriteLock**](http://docs.google.com/java/util/concurrent/locks/ReadWriteLock.html#writeLock()) Returns the lock used for writing.

**Specified by:**[writeLock](http://docs.google.com/java/util/concurrent/locks/ReadWriteLock.html#writeLock()) in interface [ReadWriteLock](http://docs.google.com/java/util/concurrent/locks/ReadWriteLock.html) **Returns:**the lock used for writing.

### readLock

public [ReentrantReadWriteLock.ReadLock](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.ReadLock.html) **readLock**()

**Description copied from interface:** [**ReadWriteLock**](http://docs.google.com/java/util/concurrent/locks/ReadWriteLock.html#readLock()) Returns the lock used for reading.

**Specified by:**[readLock](http://docs.google.com/java/util/concurrent/locks/ReadWriteLock.html#readLock()) in interface [ReadWriteLock](http://docs.google.com/java/util/concurrent/locks/ReadWriteLock.html) **Returns:**the lock used for reading.

### isFair

public final boolean **isFair**()

Returns true if this lock has fairness set true.

**Returns:**true if this lock has fairness set true

### getOwner

protected [Thread](http://docs.google.com/java/lang/Thread.html) **getOwner**()

Returns the thread that currently owns the write lock, or null if not owned. When this method is called by a thread that is not the owner, the return value reflects a best-effort approximation of current lock status. For example, the owner may be momentarily null even if there are threads trying to acquire the lock but have not yet done so. This method is designed to facilitate construction of subclasses that provide more extensive lock monitoring facilities.

**Returns:**the owner, or null if not owned

### getReadLockCount

public int **getReadLockCount**()

Queries the number of read locks held for this lock. This method is designed for use in monitoring system state, not for synchronization control.

**Returns:**the number of read locks held.

### isWriteLocked

public boolean **isWriteLocked**()

Queries if the write lock is held by any thread. This method is designed for use in monitoring system state, not for synchronization control.

**Returns:**true if any thread holds the write lock and false otherwise

### isWriteLockedByCurrentThread

public boolean **isWriteLockedByCurrentThread**()

Queries if the write lock is held by the current thread.

**Returns:**true if the current thread holds the write lock and false otherwise

### getWriteHoldCount

public int **getWriteHoldCount**()

Queries the number of reentrant write holds on this lock by the current thread. A writer thread has a hold on a lock for each lock action that is not matched by an unlock action.

**Returns:**the number of holds on the write lock by the current thread, or zero if the write lock is not held by the current thread

### getReadHoldCount

public int **getReadHoldCount**()

Queries the number of reentrant read holds on this lock by the current thread. A reader thread has a hold on a lock for each lock action that is not matched by an unlock action.

**Returns:**the number of holds on the read lock by the current thread, or zero if the read lock is not held by the current thread**Since:** 1.6

### getQueuedWriterThreads

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

Returns a collection containing threads that may be waiting to acquire the write lock. 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 lock monitoring facilities.

**Returns:**the collection of threads

### getQueuedReaderThreads

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

Returns a collection containing threads that may be waiting to acquire the read lock. 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 lock monitoring facilities.

**Returns:**the collection of threads

### hasQueuedThreads

public final boolean **hasQueuedThreads**()

Queries whether any threads are waiting to acquire the read or write lock. Note that because cancellations may occur at any time, a true return does not guarantee that any other thread will ever acquire a lock. This method is designed primarily for use in monitoring of the system state.

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

### hasQueuedThread

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

Queries whether the given thread is waiting to acquire either the read or write lock. Note that because cancellations may occur at any time, a true return does not guarantee that this thread will ever acquire a lock. This method is designed primarily for use in monitoring of the system state.

**Parameters:**thread - the thread **Returns:**true if the given thread is queued waiting for this lock **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 either the read or write lock. 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 of the system state, not for synchronization control.

**Returns:**the estimated number of threads waiting for this lock

### getQueuedThreads

protected [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 either the read or write lock. 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

### hasWaiters

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

Queries whether any threads are waiting on the given condition associated with the write lock. 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 this lock is not held [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if the given condition is not associated with this lock [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if the condition is null

### getWaitQueueLength

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

Returns an estimate of the number of threads waiting on the given condition associated with the write lock. 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 this lock is not held [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if the given condition is not associated with this lock [NullPointerException](http://docs.google.com/java/lang/NullPointerException.html) - if the condition is null

### getWaitingThreads

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

Returns a collection containing those threads that may be waiting on the given condition associated with the write lock. 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 condition monitoring facilities.

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

### toString

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

Returns a string identifying this lock, as well as its lock state. The state, in brackets, includes the String "Write locks =" followed by the number of reentrantly held write locks, and the String "Read locks =" followed by the number of held read locks.

**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 lock, as well as its lock state

| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/ReentrantReadWriteLock.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/ReentrantLock.html)   [**NEXT CLASS**](http://docs.google.com/java/util/concurrent/locks/ReentrantReadWriteLock.ReadLock.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/locks/ReentrantReadWriteLock.html)    [**NO FRAMES**](http://docs.google.com/ReentrantReadWriteLock.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |
| SUMMARY: [NESTED](#3znysh7) | FIELD | [CONSTR](#2et92p0) | [METHOD](#tyjcwt) | DETAIL: FIELD | [CONSTR](#1t3h5sf) | [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).