What is reentrant lock

Competitor APIs eventually introduce explicit locks that enrich the chances of running up to Java 4. In fact, the only way for this version of Java to be keywordd to synchronize instruction blocks, at least in the standard elements provided by JDK. The classes and interfaces provided in this part are not all locks any more. Against that said, they all control the competition for access to the blocks, so we grouped them here. One of the first criticisms of the Java synchronization model is that synchronous block entries and outputs can only be done in the same method. To solve this problem, you have a lock interface and its implementation. Lock instances exist to control access to variables shared by multiple threads at once. Variables controlled by a Lock instance can change threads with that lock one at a time. Memory management follows the same rules only if you enter and exit a block protected by a lock instance, or if you start and exit a normal synchronization block. All variables modified by the thread that locks are automatically displayed immediately by all other threads with views of these variables. The lock interface provides an overview of the following methods: lock: Try to get this lock. This method makes a hand when the lock is acquired. Therefore, if another thread has that lock, the current thread waits for the other thread to return it. Lock interrupt severability (): Attempts to retrieve this lock unless the current thread is in the INTERRUPTED state. tryLock() and tryLock (long time, in time units): Attempt to obtain this lock. If it is immediately available, these two methods return true. Otherwise, the first value immediately returns false, and the second value waits for the time indicated by the parameter that it is released. Unlock:: Release this lock. This method must be called in all cases. Therefore, this method must be called at the end of the clause. This interface is not a lock extension. readLock(): Returns a lock to playback: Provides an overview of the two methods. writeLock(): Returns a lock on the write. The two locks, read and write, are lock instances. Their theory of meaning is a little special. ReadWriteLock instances can provide two types of locks: play lock and write lock. Unless a handwriting lock is requested, this instance can provide as many playback locks as necessary. Other handWriting is exclusive: only one thread can request one. This type of locking is effective when writing is rare compared to readings. Readings can be done in conflict with access without problems, but only entries must be synchronized. I saw that sync blocks can only run one thread at a time. The thread ignores certain cases with locks and trys to enter a synchronization block with the same object settings as the one with the lock. This case often occurs, as in the following example: In this example, if you call method vogue() in a Captain instance, the fashion method of the Marin class is also called. These two methods are synchronized to the same object. So what's going on? The thread has a lock and trys to enter a synchronization block where the lock is not available, but the thread turns out to own the lock. Java sync is said to be re-insible, in which case a thread with the appropriate lock can enter the new sync block. For Lock instances, this point is not specified, so it is the responsibility of the implementation class to determine whether it is re-in. The implementations provided by the standard API are: The ReturningLock class, as its name implies, is an implementation of lock reenscible characters. This publishes some additional methods compared to the lock interface, which is very useful. getHoldCount(): Returns the number of times the lock method () of this lock was called without unlocking (method) being called. Keep in mind that in a lock call, () requires a call to unlock(), and this lock must only be opened if there are many calls to these two methods. getQueueLength():Returns the number of pending threads in this lock. Returns true if the thread is waiting for this lock, or if the configured thread is in that queue. Returns true if the current thread is waiting for this lock. isLocked(): Returns true if this lock is retrieved by a thread. Let's take a look at an example of how to use this class. Classes implement interface semantics. Similar to ReentrantLock, it provides an overview of several methods for examining threads waiting for locks. Because read locks are different from write locks, the re-in mechanism is slightly different. Read lock is re-enterable withoutSe Cui Est Naturerian Thread Quy Possed un Verrow de Lecture ne Pu Pa Obtenil de Verow Decricher-sur-Memm Objet Instance de RentrentrantLock Il Doit Tote Dabold Reveller Son Verou de Lecture Anne Levanche, Anne Thread Cui Possed Un Verow Decricher Pew Akkeril un Vel de Lecture sur Meme Objet, Puy Liberer son Verow Decricher. java.lang.Object java.util.concrete.locks.ReentrantLock All implemented interfaces: serializable, extended object implementation of lock public class ReentrantLock, serializable A-reentrant mutual exclusive lock with the same basic behavior and semantics as implicit monitor locks accessed using synchronized methods and statements. ReentrantLock is owned by the last thread that successfully locked it, but it has not yet been unlocked. The thread that called the lock successfully gets the lock if it is not owned by another thread. If the current thread already owns the lock, the method is returned immediately. This can be checked using the method. Constructors of this class accept optional fairness parameters. If set to true, under contention, the lock prioritizes allowing access to the longest waiting thread. Otherwise, this lock does not guarantee a specific order of access. Programs that use fairlocks accessed by many threads may appear to have lower overall throughput than programs that use the default settings (that is, they are usually slower, but have a smaller time difference in time to obtain locks and guarantee a lack of starvation). However, lock fairness does not guarantee thread scheduling fairness. Therefore, one of many threads that use fair locks may get multiple times in a row, while the other active threads are not progressing and are not currently holding locks. Also note that the no-timed tryLock method does not respect the fairness setting. This process succeeds if a lock is available, even if other threads are waiting. We recommend that you always follow the calls you lock in the try block. Public void m() { lock.lock(); // Block until the condition holds the attempt { // .method body } Finally { lock.unlock() } } In addition to implementing the lock interface, this class defines methods isLocked and getLockQueueLength, and several associated protected access methods that may be useful for instrumentation and monitoring. Serialization of this class works in the same way as built-in locks. This lock supports up to 2147483647 recursed locks by the same thread. If you try to exceed this limit, Error is thrown from the lock method. Since: 1.5 See: Serialized FormsCreates an instance of a re-enter lock. This is the same as using false. Public ReentrentLock Creates an instance of a reentry lock using the specified fairness policy. Parameter: fair - gets a true public void lock() lock if this lock must use a fair ordering policy. Gets the lock if it is not held by another thread, immediately returns control, and set the lock retention count to 1. If the current thread already holds the lock, the hold count is incremented one at a time and the method returns immediately. If the lock is held by another thread, the current thread becomes unavailable for thread scheduling, hibernates until the lock is won, and at that point the lock hold count is set to 1. Specified: Interface lock public void lockInterruptibly() Slow interrupt exception Gets the lock unless the current thread is interrupted. Gets the lock if it is not held by another thread, immediately returns control, and set the lock retention count to 1. If the current thread already holds this lock, the hold count is incremented one by one, and the method returns immediately. If the lock is held by another thread, the current thread is disabled for thread scheduling and hibernates until one of the following two operations occurs: Alternatively, another thread suspends the current thread. If the lock is retrieved by the current thread, the lock retention count is set to 1. Current thread: if has an interrupt state set in the entry to this method. If interrupted while retrieving a lock, InterruptedException is thrown to clear the suspended state of the current thread. In this implementation, because this method is an explicit break point, it is preferred to respond to interrupts over normal or re-enterable acquisition of locks. Specified: lockInterruptibly Interface Lock Throw: InterruptedException - If the current thread is interrupted, the public Boolean value tryLock() gets the lock only if it is not held by another thread at the time of the call. Gets the lock if the lock is not held by another thread, immediately returns the value true, and set the lock retention count to 1. Even if this lock is set to use a fair ordering policy, the tryLock() call gets the lock immediately if the lock is available, regardless of whether another thread is currently waiting for the lock. This interrupt behavior is useful in certain situations, even if it breaks fairness. If you want to respect this lock's fairness setting, use a roughly equivalent tryLock (0, TimeUnit.SECONDS) (it also detects interruptions). If the current thread already holds this lock, the hold count is incremented one by one, and the method returns true. If the lock is held by another thread, this methodThe value is false. Specified: Return lock lock on tryLock interface: true If the lock is released and retrieved by the current thread, or if the lock was already held by the current thread. false Otherwise, public boules tryLock (long timeout, TimeUnit unit) throws an interrupt exception when it is not held by another thread within the specified wait time and the current thread is not interrupted. Gets the lock if the lock is not held by another thread, immediately returns the value true, and set the lock retention count to 1. If this lock is set to use a fair ordering policy, no available locks are gained if other threads are waiting for the lock. This is in contrast to the tryLock() method. If you need a timed tryLock to interrupt with fairlocks, combine timed and non-timing forms. If the current thread already holds this lock, the hold count is incremented one by one, and the method returns true. If the lock is held by another thread, the current thread is disabled for thread scheduling and hibernates until one of the following three occurs: Or another thread interrupts the current thread. Or, if a lock with a specified wait time is retrieved, the value true is returned and the lock retention count is set to 1. Current thread: if has an interrupt state set in the entry to this method. If interrupted while retrieving a lock, InterruptedException is thrown to clear the suspended state of the current thread. Returns the value false after the specified wait time. If the time is less than or less than zero, the method does not wait at all. Because this method is an explicit break point, this implementation prioritizes responding to interrupts for normal or re-intible acquisition of locks and reporting the progress of latency. Specified: tryLock in-interface lock parameter: Timeout - Time to wait for lock unit - Time unit of timeout argument returns if the lock is free and retrieved by the current thread, or if the lock was already held by the current thread. Also, false Throws: InterruptedException if the wait time has elapsed before the lock is acquired - NullPointerException if the current thread is interrupted - Public void unlock() with a null time unit, will try to release this lock. If the current thread is the holder of this lock, the hold count decreases. If the hold count is zero, the lock is released. If the current thread is not the owner of this lock, an invalid monitor state exception is thrown. Specified: Unlock with interface lock throw: Bad monitor state exception - if the current thread does not hold itReturns the condition instance to use for this lock instance. The Condition instance returned supports the same usage as the Object Monitor method (Wait, Notify, and NotifyAll) when used with built-in monitor locks. If this lock is not held when either the condition wait method or the signal method is called, a bad monitor state exception is thrown. When the condition wait method is called, the lock is released, the lock is re-acquired before it returns, and the lock hold count is restored to the state it was in when the method was called. If a thread is interrupted while waiting, the wait ends, interruptedException is thrown, and the thread's suspended state is cleared. Waiting threads are signaled in FIFO order. The order in which locks are re-retrieved for threads returning from the wait method is the same as for the thread that gets the lock first, which is not specified in the default case, but for fair locks, it takes precedence over the thread that has been waiting the longest. Specified: NewCondition interface lock return value: Condition object public int getHoldCount() Queries the number of holds for this lock by the current thread. The thread holds the lock for each lock action that does not match in the unlock action. Hold count information is typically used only for testing and debugging purposes. For example, if you don't need to enter a specific section of code with a lock that already holds it, you can assert that fact. public void m() { assert lock.getHoldCount() == 0; lock.lock(); { // .Method Body } Last { Unlock() } } }} Return: The number of holds for this lock by the current thread, or 0 if this lock is not held by the current thread, if this lock is held by the current thread, if this lock is held, the query is held by the current thread. Similar to the Thread.holdsLock (java.lang.Object) method of the built-in monitor lock, but this method is typically used for debugging and testing. For example, a method that should only be called while the lock is held can assert that this is the case: class X { ReentLock lock = new ReentLock(); // .public void m() { assert lock.isHeldByCurrentThread();// .. Method body } } Can also be used to ensure that reentrant locks are used in an irrevocable way. Public void m() { Assert !lock.isHeldByCurrentThread(); lock.lock();method body } finally { unlock() } } }} Returns true if the current thread holds this lock, otherwise the public boules value returns an isLocked() query (if this lock is held by one of the threads). This method is designed to be used for monitoring system conditions, not synchronous control. Returns true if one of the threads holds this lock;Returns true if fairness is set to true for this lock. True If this lock is set fairly, the protected Thread getOwner() returns the thread that currently owns this lock. When this method is called by a thread that is not the owner, the return value reflects an approximation of the best effort of the current lock state. For example, the owner might be temporarily null even if the thread trying to retrieve the lock has not yet acquired the lock. This method is designed to facilitate the construction of subclasses that provide broader lock monitoring capabilities. Return: If you do not own the owner, or the final Boolean value of the null public, null queries whether the thread is waiting to acquire this lock. Returning the actual return value does not guarantee that other threads will win this lock, as cancellations can occur at any time. This method is primarily designed for use in monitoring system conditions. Asks if the specified thread is waiting to acquire this lock, and returns any other threads waiting to get the public final Boolean value of the lock. Returning the actual return value does not guarantee that this thread will win this lock, as cancellations can occur at any time. This method is primarily designed for use in monitoring system conditions. Parameters:Thread - The thread returns if the specified thread is queued waiting for this lock:true Throw: NullPointerException - If the thread is null, it returns an estimate of the number of threads waiting to get this lock. The value is only an estimate because the number of threads can change dynamically while this method traverses the internal data structure. This method is designed to be used for monitoring system conditions, not synchronous control. Return: The estimated number of threads waiting for this lock protection collection getQueuedThreads() &lt;Thread&gt; returns a collection containing threads that may be waiting to get this lock. The actual set of threads can change dynamically as you build this result, so the collection returned is only a best effort estimate. The elements in the collection returned are never ordered. This method is designed to facilitate the construction of subclasses that provide broader monitoring capabilities. Return: Public Boolean value of the thread hasWaiters (condition) Queries whether there is a thread waiting for a specific condition associated with this lock. True returns do not guarantee that future signals are calling threads, as timeouts and interrupts can occur at any time. This method is primarily designed for use in monitoring system conditions. Parameter: Condition - The condition returns if there is a wait thread:true Throw: &lt;/Thread&gt;If the specified condition is not associated with this lock NullPointerException, getWaitQueueLength (condition) returns an estimate of the number of threads waiting for a particular condition associated with this lock if the specified condition is null. Because timeouts and interrupts can occur at any time, the estimate only works as a limit on the actual number of waits. This method is designed to be used for monitoring system conditions, not synchronous control. Parameters: Condition - Condition Return: Estimated Number of Wait threads Throw: IllegalMonitorStateException - NullArgumentException if this lock is not held - NullPointerException where the specified condition is this lock If not associated with - the condition returns a null-protected collection &lt;Thread&gt;getWaitingThreads(condition) that contains a thread that may be waiting for a specific condition associated with this lock. The actual set of threads can change dynamically as you build this result, so the collection returned is only a best effort estimate. The elements in the collection returned are never ordered. This method is designed to facilitate the construction of subclasses that provide broader condition monitoring capabilities. Parameters: Condition - Condition Return: Thread Collection Throw: IllegalMonitorStateException - NullPointerException if this lock is not held - If the specified condition is not associated with this lock NullPointerException - Returns a string and lock state that identifies this lock if the condition is null public String to String(). The state is enclosed in brackets, followed by a string-locked or locked string followed by the name of the owning thread. Override: class object return toString: See the Java SE documentation for strings that identify this lock, as well as lock state bugs and feature submissions, as well as other API references and developer documentation. This document contains a more detailed developer description that includes a conceptual overview, term definitions, workarounds, and working code examples. Copyright © 1993, 2020, Oracle and its affiliates. All rights are reserved. Use is in accordance with the license terms. See also document redistribution policies. Policy. &lt;/Thread&gt;