<http://tutorials.jenkov.com/java-util-concurrent/readwritelock.html>

# ReadWriteLock

* [ReadWriteLock Locking Rules](http://tutorials.jenkov.com/java-util-concurrent/readwritelock.html#read-write-lock-locking-rules)
* [ReadWriteLock Implementations](http://tutorials.jenkov.com/java-util-concurrent/readwritelock.html#read-write-lock-implementations)
* [ReadWriteLock Code Example](http://tutorials.jenkov.com/java-util-concurrent/readwritelock.html#read-write-lock-code-example)

A java.util.concurrent.locks.ReadWriteLock is an advanced thread lock mechanism. It allows multiple threads to read a certain resource, but only one to write it, at a time.

The idea is, that multiple threads can read from a shared resource without causing concurrency errors. The concurrency errors first occur when reads and writes to a shared resource occur concurrently, or if multiple writes take place concurrently.

In this text I only cover Java's built-in ReadWriteLock. If you want to read more about the theory behind the implemenation of a ReadWriteLock, you can read it in my text on [**Read Write Locks**](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html) in my Java Concurrency tutorial.

## ReadWriteLock Locking Rules

The rules by which a thread is allowed to lock the ReadWriteLock either for reading or writing the guarded resource, are as follows:

|  |  |
| --- | --- |
| **Read Lock** | If no threads have locked the ReadWriteLock for writing, and no thread have requested a write lock (but not yet obtained it). Thus, multiple threads can lock the lock for reading. |
| **Write Lock** | If no threads are reading or writing. Thus, only one thread at a time can lock the lock for writing. |

## ReadWriteLock Implementations

ReadWriteLock is an interface. Thus, to use a ReadWriteLock

The java.util.concurrent.locks package contains the following ReadWriteLock implementation:

* ReentrantReadWriteLock

## ReadWriteLock Code Example

Here is a simple code example that shows how to create a ReadWriteLock and how to lock it for reading and writing:

ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

readWriteLock.readLock().lock();

// multiple readers can enter this section

// if not locked for writing, and not writers waiting

// to lock for writing.

readWriteLock.readLock().unlock();

readWriteLock.writeLock().lock();

// only one writer can enter this section,

// and only if no threads are currently reading.

readWriteLock.writeLock().unlock();

Notice how the ReadWriteLock actually internally keeps two Lock instances. One guarding read access, and one guarding write access.

<http://tutorials.jenkov.com/java-concurrency/read-write-locks.html>

# Read / Write Locks in Java

* [Read / Write Lock Java Implementation](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#simple)
* [Read / Write Lock Reentrance](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#reentrance)
* [Read Reentrance](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#readreentrance)
* [Write Reentrance](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#writereentrance)
* [Read to Write Reentrance](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#upgrade)
* [Write to Read Reentrance](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#downgrade)
* [Fully Reentrant ReadWriteLock](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#full)
* [Calling unlock() From a finally-clause](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html#finally)

A read / write lock is more sophisticated lock than the Lock implementations shown in the text [**Locks in Java**](http://tutorials.jenkov.com/java-concurrency/locks.html). Imagine you have an application that reads and writes some resource, but writing it is not done as much as reading it is. Two threads reading the same resource does not cause problems for each other, so multiple threads that want to read the resource are granted access at the same time, overlapping. But, if a single thread wants to write to the resource, no other reads nor writes must be in progress at the same time. To solve this problem of allowing multiple readers but only one writer, you will need a read / write lock.

Java 5 comes with read / write lock implementations in the java.util.concurrent package. Even so, it may still be useful to know the theory behind their implementation.

## Read / Write Lock Java Implementation

First let's summarize the conditions for getting read and write access to the resource:

|  |  |
| --- | --- |
| **Read Access** | If no threads are writing, and no threads have requested write access. |
| **Write Access** | If no threads are reading or writing. |

If a thread wants to read the resource, it is okay as long as no threads are writing to it, and no threads have requested write access to the resource. By up-prioritizing write-access requests we assume that write requests are more important than read-requests. Besides, if reads are what happens most often, and we did not up-prioritize writes, [**starvation**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html) could occur. Threads requesting write access would be blocked until all readers had unlocked the ReadWriteLock. If new threads were constantly granted read access the thread waiting for write access would remain blocked indefinately, resulting in [**starvation**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html). Therefore a thread can only be granted read access if no thread has currently locked the ReadWriteLock for writing, or requested it locked for writing.

A thread that wants write access to the resource can be granted so when no threads are reading nor writing to the resource. It doesn't matter how many threads have requested write access or in what sequence, unless you want to guarantee fairness between threads requesting write access.

With these simple rules in mind we can implement a ReadWriteLock as shown below:

public class ReadWriteLock{

private int readers = 0;

private int writers = 0;

private int writeRequests = 0;

public synchronized void lockRead() throws InterruptedException{

while(writers > 0 || writeRequests > 0){

wait();

}

readers++;

}

public synchronized void unlockRead(){

readers--;

notifyAll();

}

public synchronized void lockWrite() throws InterruptedException{

writeRequests++;

while(readers > 0 || writers > 0){

wait();

}

writeRequests--;

writers++;

}

public synchronized void unlockWrite() throws InterruptedException{

writers--;

notifyAll();

}

}

The ReadWriteLock has two lock methods and two unlock methods. One lock and unlock method for read access and one lock and unlock for write access.

The rules for read access are implemented in the lockRead() method. All threads get read access unless there is a thread with write access, or one or more threads have requested write access.

The rules for write access are implemented in the lockWrite() method. A thread that wants write access starts out by requesting write access (writeRequests++). Then it will check if it can actually get write access. A thread can get write access if there are no threads with read access to the resource, and no threads with write access to the resource. How many threads have requested write access doesn't matter.

It is worth noting that both unlockRead() and unlockWrite() calls notifyAll() rather than notify(). To explain why that is, imagine the following situation:

Inside the ReadWriteLock there are threads waiting for read access, and threads waiting for write access. If a thread awakened by notify() was a read access thread, it would be put back to waiting because there are threads waiting for write access. However, none of the threads awaiting write access are awakened, so nothing more happens. No threads gain neither read nor write access. By calling noftifyAll() all waiting threads are awakened and check if they can get the desired access.

Calling notifyAll() also has another advantage. If multiple threads are waiting for read access and none for write access, and unlockWrite() is called, all threads waiting for read access are granted read access at once - not one by one.

## Read / Write Lock Reentrance

The ReadWriteLock class shown earlier is not [**reentrant**](http://tutorials.jenkov.com/java-concurrency/locks.html#reentrance). If a thread that has write access requests it again, it will block because there is already one writer - itself. Furthermore, consider this case:

1. Thread 1 gets read access.
2. Thread 2 requests write access but is blocked because there is one reader.
3. Thread 1 re-requests read access (re-enters the lock), but is blocked because there is a write request

In this situation the previous ReadWriteLock would lock up - a situation similar to deadlock. No threads requesting neither read nor write access would be granted so.

To make the ReadWriteLock reentrant it is necessary to make a few changes. Reentrance for readers and writers will be dealt with separately.

## Read Reentrance

To make the ReadWriteLock reentrant for readers we will first establish the rules for read reentrance:

* A thread is granted read reentrance if it can get read access (no writers or write requests), or if it already has read access (regardless of write requests).

To determine if a thread has read access already a reference to each thread granted read access is kept in a Map along with how many times it has acquired read lock. When determing if read access can be granted this Map will be checked for a reference to the calling thread. Here is how the lockRead() and unlockRead() methods looks after that change:

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writers = 0;

private int writeRequests = 0;

public synchronized void lockRead() throws InterruptedException{

Thread callingThread = Thread.currentThread();

while(! canGrantReadAccess(callingThread)){

wait();

}

readingThreads.put(callingThread,

(getAccessCount(callingThread) + 1));

}

public synchronized void unlockRead(){

Thread callingThread = Thread.currentThread();

int accessCount = getAccessCount(callingThread);

if(accessCount == 1){ readingThreads.remove(callingThread); }

else { readingThreads.put(callingThread, (accessCount -1)); }

notifyAll();

}

private boolean canGrantReadAccess(Thread callingThread){

if(writers > 0) return false;

if(isReader(callingThread) return true;

if(writeRequests > 0) return false;

return true;

}

private int getReadAccessCount(Thread callingThread){

Integer accessCount = readingThreads.get(callingThread);

if(accessCount == null) return 0;

return accessCount.intValue();

}

private boolean isReader(Thread callingThread){

return readingThreads.get(callingThread) != null;

}

}

As you can see read reentrance is only granted if no threads are currently writing to the resource. Additionally, if the calling thread already has read access this takes precedence over any writeRequests.

## Write Reentrance

Write reentrance is granted only if the thread has already write access. Here is how the lockWrite() and unlockWrite() methods look after that change:

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writeAccesses = 0;

private int writeRequests = 0;

private Thread writingThread = null;

public synchronized void lockWrite() throws InterruptedException{

writeRequests++;

Thread callingThread = Thread.currentThread();

while(! canGrantWriteAccess(callingThread)){

wait();

}

writeRequests--;

writeAccesses++;

writingThread = callingThread;

}

public synchronized void unlockWrite() throws InterruptedException{

writeAccesses--;

if(writeAccesses == 0){

writingThread = null;

}

notifyAll();

}

private boolean canGrantWriteAccess(Thread callingThread){

if(hasReaders()) return false;

if(writingThread == null) return true;

if(!isWriter(callingThread)) return false;

return true;

}

private boolean hasReaders(){

return readingThreads.size() > 0;

}

private boolean isWriter(Thread callingThread){

return writingThread == callingThread;

}

}

Notice how the thread currently holding the write lock is now taken into account when determining if the calling thread can get write access.

## Read to Write Reentrance

Sometimes it is necessary for a thread that have read access to also obtain write access. For this to be allowed the thread must be the only reader. To achieve this the writeLock() method should be changed a bit. Here is what it would look like:

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writeAccesses = 0;

private int writeRequests = 0;

private Thread writingThread = null;

public synchronized void lockWrite() throws InterruptedException{

writeRequests++;

Thread callingThread = Thread.currentThread();

while(! canGrantWriteAccess(callingThread)){

wait();

}

writeRequests--;

writeAccesses++;

writingThread = callingThread;

}

public synchronized void unlockWrite() throws InterruptedException{

writeAccesses--;

if(writeAccesses == 0){

writingThread = null;

}

notifyAll();

}

private boolean canGrantWriteAccess(Thread callingThread){

**if(isOnlyReader(callingThread)) return true;**

if(hasReaders()) return false;

if(writingThread == null) return true;

if(!isWriter(callingThread)) return false;

return true;

}

private boolean hasReaders(){

return readingThreads.size() > 0;

}

private boolean isWriter(Thread callingThread){

return writingThread == callingThread;

}

**private boolean isOnlyReader(Thread thread){**

**return readingThreads.size() == 1 &&**

**readingThreads.get(callingThread) != null;**

**}**

}

Now the ReadWriteLock class is read-to-write access reentrant.

## Write to Read Reentrance

Sometimes a thread that has write access needs read access too. A writer should always be granted read access if requested. If a thread has write access no other threads can have read nor write access, so it is not dangerous. Here is how the canGrantReadAccess() method will look with that change:

public class ReadWriteLock{

private boolean canGrantReadAccess(Thread callingThread){

**if(isWriter(callingThread)) return true;**

if(writingThread != null) return false;

if(isReader(callingThread) return true;

if(writeRequests > 0) return false;

return true;

}

}

## Fully Reentrant ReadWriteLock

Below is the fully reentran ReadWriteLock implementation. I have made a few refactorings to the access conditions to make them easier to read, and thereby easier to convince yourself that they are correct.

public class ReadWriteLock{

private Map<Thread, Integer> readingThreads =

new HashMap<Thread, Integer>();

private int writeAccesses = 0;

private int writeRequests = 0;

private Thread writingThread = null;

public synchronized void lockRead() throws InterruptedException{

Thread callingThread = Thread.currentThread();

while(! canGrantReadAccess(callingThread)){

wait();

}

readingThreads.put(callingThread,

(getReadAccessCount(callingThread) + 1));

}

private boolean canGrantReadAccess(Thread callingThread){

if( isWriter(callingThread) ) return true;

if( hasWriter() ) return false;

if( isReader(callingThread) ) return true;

if( hasWriteRequests() ) return false;

return true;

}

public synchronized void unlockRead(){

Thread callingThread = Thread.currentThread();

if(!isReader(callingThread)){

throw new IllegalMonitorStateException("Calling Thread does not" +

" hold a read lock on this ReadWriteLock");

}

int accessCount = getReadAccessCount(callingThread);

if(accessCount == 1){ readingThreads.remove(callingThread); }

else { readingThreads.put(callingThread, (accessCount -1)); }

notifyAll();

}

public synchronized void lockWrite() throws InterruptedException{

writeRequests++;

Thread callingThread = Thread.currentThread();

while(! canGrantWriteAccess(callingThread)){

wait();

}

writeRequests--;

writeAccesses++;

writingThread = callingThread;

}

public synchronized void unlockWrite() throws InterruptedException{

if(!isWriter(Thread.currentThread()){

throw new IllegalMonitorStateException("Calling Thread does not" +

" hold the write lock on this ReadWriteLock");

}

writeAccesses--;

if(writeAccesses == 0){

writingThread = null;

}

notifyAll();

}

private boolean canGrantWriteAccess(Thread callingThread){

if(isOnlyReader(callingThread)) return true;

if(hasReaders()) return false;

if(writingThread == null) return true;

if(!isWriter(callingThread)) return false;

return true;

}

private int getReadAccessCount(Thread callingThread){

Integer accessCount = readingThreads.get(callingThread);

if(accessCount == null) return 0;

return accessCount.intValue();

}

private boolean hasReaders(){

return readingThreads.size() > 0;

}

private boolean isReader(Thread callingThread){

return readingThreads.get(callingThread) != null;

}

private boolean isOnlyReader(Thread callingThread){

return readingThreads.size() == 1 &&

readingThreads.get(callingThread) != null;

}

private boolean hasWriter(){

return writingThread != null;

}

private boolean isWriter(Thread callingThread){

return writingThread == callingThread;

}

private boolean hasWriteRequests(){

return this.writeRequests > 0;

}

}

## Calling unlock() From a finally-clause

When guarding a critical section with a ReadWriteLock, and the critical section may throw exceptions, it is important to call the readUnlock() and writeUnlock() methods from inside a finally-clause. Doing so makes sure that the ReadWriteLock is unlocked so other threads can lock it. Here is an example:

lock.lockWrite();

try{

//do critical section code, which may throw exception

} finally {

lock.unlockWrite();

}

This little construct makes sure that the ReadWriteLock is unlocked in case an exception is thrown from the code in the critical section. If unlockWrite() was not called from inside a finally-clause, and an exception was thrown from the critical section, the ReadWriteLock would remain write locked forever, causing all threads calling lockRead() or lockWrite() on that ReadWriteLock instance to halt indefinately. The only thing that could unlock the ReadWriteLockagain would be if the ReadWriteLock is reentrant, and the thread that had it locked when the exception was thrown, later succeeds in locking it, executing the critical section and calling unlockWrite() again afterwards. That would unlock the ReadWriteLock again. But why wait for that to happen, **if** it happens? Calling unlockWrite() from a finally-clause is a much more robust solution.