**Locks in Java**

* [A Simple Lock](http://tutorials.jenkov.com/java-concurrency/locks.html#simple-lock)
* [Lock Reentrance](http://tutorials.jenkov.com/java-concurrency/locks.html#reentrance)
* [Lock Fairness](http://tutorials.jenkov.com/java-concurrency/locks.html#fairness)
* [Calling unlock() From a finally-clause](http://tutorials.jenkov.com/java-concurrency/locks.html#finally)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2014-06-23 |

A lock is a thread synchronization mechanism like synchronized blocks except locks can be more sophisticated than Java's synchronized blocks. Locks (and other more advanced synchronization mechanisms) are created using synchronized blocks, so it is not like we can get totally rid of thesynchronized keyword.

From Java 5 the package java.util.concurrent.locks contains several lock implementations, so you may not have to implement your own locks. But you will still need to know how to use them, and it can still be useful to know the theory behind their implementation. For more details, see my tutorial on the[**java.util.concurrent.locks.Lock**](http://tutorials.jenkov.com/java-util-concurrent/lock.html) interface.

**A Simple Lock**

Let's start out by looking at a synchronized block of Java code:

public class Counter{

private int count = 0;

public int inc(){

synchronized(this){

return ++count;

}

}

}

Notice the synchronized(this) block in the inc() method. This block makes sure that only one thread can execute the return ++count at a time. The code in the synchronized block could have been more advanced, but the simple ++count suffices to get the point across.

The Counter class could have been written like this instead, using a Lock instead of a synchronized block:

public class Counter{

private Lock lock = new Lock();

private int count = 0;

public int inc(){

lock.lock();

int newCount = ++count;

lock.unlock();

return newCount;

}

}

The lock() method locks the Lock instance so that all threads calling lock() are blocked until unlock() is executed.

Here is a simple Lock implementation:

public class Lock{

private boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

}

public synchronized void unlock(){

isLocked = false;

notify();

}

}

Notice the while(isLocked) loop, which is also called a "spin lock". Spin locks and the methods wait() andnotify() are covered in more detail in the text [**Thread Signaling**](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html). While isLocked is true, the thread callinglock() is parked waiting in the wait() call. In case the thread should return unexpectedly from the wait() call without having received a notify() call (AKA a [**Spurious Wakeup**](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#spuriouswakeups)) the thread re-checks the isLockedcondition to see if it is safe to proceed or not, rather than just assume that being awakened means it is safe to proceed. If isLocked is false, the thread exits the while(isLocked) loop, and sets isLocked back to true, to lock the Lock instance for other threads calling lock().

When the thread is done with the code in the [**critical section**](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html) (the code between lock() and unlock()), the thread calls unlock(). Executing unlock() sets isLocked back to false, and notifies (awakens) one of the threads waiting in the wait() call in the lock() method, if any.

**Lock Reentrance**

Synchronized blocks in Java are reentrant. This means, that if a Java thread enters a synchronized block of code, and thereby take the lock on the monitor object the block is synchronized on, the thread can enter other Java code blocks synchronized on the same monitor object. Here is an example:

public class Reentrant{

public synchronized outer(){

inner();

}

public synchronized inner(){

//do something

}

}

Notice how both outer() and inner() are declared synchronized, which in Java is equivalent to asynchronized(this) block. If a thread calls outer() there is no problem calling inner() from inside outer(), since both methods (or blocks) are synchronized on the same monitor object ("this"). If a thread already holds the lock on a monitor object, it has access to all blocks synchronized on the same monitor object. This is called reentrance. The thread can reenter any block of code for which it already holds the lock.

The lock implementation shown earlier is not reentrant. If we rewrite the Reentrant class like below, the thread calling outer() will be blocked inside the lock.lock() in the inner() method.

public class Reentrant2{

Lock lock = new Lock();

public outer(){

lock.lock();

inner();

lock.unlock();

}

public synchronized inner(){

lock.lock();

//do something

lock.unlock();

}

}

A thread calling outer() will first lock the Lock instance. Then it will call inner(). Inside the inner() method the thread will again try to lock the Lock instance. This will fail (meaning the thread will be blocked), since the Lock instance was locked already in the outer() method.

The reason the thread will be blocked the second time it calls lock() without having called unlock() in between, is apparent when we look at the lock() implementation:

public class Lock{

boolean isLocked = false;

public synchronized void lock()

throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

}

...

}

It is the condition inside the while loop (spin lock) that determines if a thread is allowed to exit the lock()method or not. Currently the condition is that isLocked must be false for this to be allowed, regardless of what thread locked it.

To make the Lock class reentrant we need to make a small change:

public class Lock{

boolean isLocked = false;

Thread lockedBy = null;

int lockedCount = 0;

public synchronized void lock()

throws InterruptedException{

Thread callingThread = Thread.currentThread();

while(isLocked && lockedBy != callingThread){

wait();

}

isLocked = true;

lockedCount++;

lockedBy = callingThread;

}

public synchronized void unlock(){

if(Thread.curentThread() == this.lockedBy){

lockedCount--;

if(lockedCount == 0){

isLocked = false;

notify();

}

}

}

...

}

Notice how the while loop (spin lock) now also takes the thread that locked the Lock instance into consideration. If either the lock is unlocked (isLocked = false) or the calling thread is the thread that locked the Lock instance, the while loop will not execute, and the thread calling lock() will be allowed to exit the method.

Additionally, we need to count the number of times the lock has been locked by the same thread. Otherwise, a single call to unlock() will unlock the lock, even if the lock has been locked multiple times. We don't want the lock to be unlocked until the thread that locked it, has executed the same amount ofunlock() calls as lock() calls.

The Lock class is now reentrant.

**Lock Fairness**

Java's synchronized blocks makes no guarantees about the sequence in which threads trying to enter them are granted access. Therefore, if many threads are constantly competing for access to the same synchronized block, there is a risk that one or more of the threads are never granted access - that access is always granted to other threads. This is called starvation. To avoid this a Lock should be fair. Since theLock implementations shown in this text uses synchronized blocks internally, they do not guarantee fairness. Starvation and fairness are discussed in more detail in the text [**Starvation and Fairness**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html).

**Calling unlock() From a finally-clause**

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

lock.lock();

try{

//do critical section code, which may throw exception

} finally {

lock.unlock();

}

This little construct makes sure that the Lock is unlocked in case an exception is thrown from the code in the critical section. If unlock() was not called from inside a finally-clause, and an exception was thrown from the critical section, the Lock would remain locked forever, causing all threads calling lock() on thatLock instance to halt indefinately.