**<http://tutorials.jenkov.com/java-concurrency/volatile.html>**

**The Java synchronized Keyword**

Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

The synchronized keyword can be used to mark four different types of blocks:

1. Instance methods
2. Static methods
3. Code blocks inside instance methods
4. Code blocks inside static methods

These blocks are synchronized on different objects. Which type of synchronized block you need depends on the concrete situation.

**Synchronized Instance Methods**

Here is a synchronized instance method:

public **synchronized** void add(int value){

this.count += value;

}

Notice the use of the synchronized keyword in the method declaration. This tells Java that the method is synchronized.

A synchronized instance method in Java is synchronized on the instance (object) owning the method. Thus, each instance has its synchronized methods synchronized on a different object: the owning instance. Only one thread can execute inside a synchronized instance method. If more than one instance exist, then one thread at a time can execute inside a synchronized instance method per instance. One thread per instance.

**Synchronized Static Methods**

Static methods are marked as synchronized just like instance methods using the synchronized keyword. Here is a Java synchronized static method example:

public **static synchronized** void add(int value){

count += value;

}

Also here the synchronized keyword tells Java that the method is synchronized.

Synchronized static methods are synchronized on the class object of the class the synchronized static method belongs to. Since only one class object exists in the Java VM per class, only one thread can execute inside a static synchronized method in the same class.

If the static synchronized methods are located in different classes, then one thread can execute inside the static synchronized methods of each class. One thread per class regardless of which static synchronized method it calls.

**Synchronized Blocks in Instance Methods**

You do not have to synchronize a whole method. Sometimes it is preferable to synchronize only part of a method. Java synchronized blocks inside methods makes this possible.

Here is a synchronized block of Java code inside an unsynchronized Java method:

public void add(int value){

**synchronized(this){**

this.count += value;

**}**

}

This example uses the Java synchronized block construct to mark a block of code as synchronized. This code will now execute as if it was a synchronized method.

Notice how the Java synchronized block construct takes an object in parentheses. In the example "this" is used, which is the instance the add method is called on. The object taken in the parentheses by the synchronized construct is called a monitor object. The code is said to be synchronized on the monitor object. A synchronized instance method uses the object it belongs to as monitor object.

Only one thread can execute inside a Java code block synchronized on the same monitor object.

The following two examples are both synchronized on the instance they are called on. They are therefore equivalent with respect to synchronization:

public class MyClass {

public **synchronized** void log1(String msg1, String msg2){

log.writeln(msg1);

log.writeln(msg2);

}

public void log2(String msg1, String msg2){

**synchronized(this){**

log.writeln(msg1);

log.writeln(msg2);

**}**

}

}

Thus only a single thread can execute inside either of the two synchronized blocks in this example.

Had the second synchronized block been synchronized on a different object than this, then one thread at a time had been able to execute inside each method.

**Synchronized Blocks in Static Methods**

Here are the same two examples as static methods. These methods are synchronized on the class object of the class the methods belong to:

public class MyClass {

public static synchronized void log1(String msg1, String msg2){

log.writeln(msg1);

log.writeln(msg2);

}

public static void log2(String msg1, String msg2){

synchronized(MyClass.class){

log.writeln(msg1);

log.writeln(msg2);

}

}

}

Only one thread can execute inside any of these two methods at the same time.

Had the second synchronized block been synchronized on a different object than MyClass.class, then one thread could execute inside each method at the same time.

**Java Synchronized Example**

Here is an example that starts 2 threads and have both of them call the add method on the same instance of Counter. Only one thread at a time will be able to call the add method on the same instance, because the method is synchronized on the instance it belongs to.

public class Counter{

long count = 0;

public synchronized void add(long value){

this.count += value;

}

}

public class CounterThread extends Thread{

protected Counter counter = null;

public CounterThread(Counter counter){

this.counter = counter;

}

public void run() {

for(int i=0; i<10; i++){

counter.add(i);

}

}

}

public class Example {

public static void main(String[] args){

Counter counter = new Counter();

Thread threadA = new CounterThread(counter);

Thread threadB = new CounterThread(counter);

threadA.start();

threadB.start();

}

}

Two threads are created. The same Counter instance is passed to both of them in their constructor. TheCounter.add() method is synchronized on the instance, because the add method is an instance method, and marked as synchronized. Therefore only one of the threads can call the add() method at a time. The other thread will wait until the first thread leaves the add() method, before it can execute the method itself.

If the two threads had referenced two separate Counter instances, there would have been no problems calling the add() methods simultaneously. The calls would have been to different objects, so the methods called would also be synchronized on different objects (the object owning the method). Therefore the calls would not block. Here is how that could look:

public class Example {

public static void main(String[] args){

Counter counterA = new Counter();

Counter counterB = new Counter();

Thread threadA = new CounterThread(counterA);

Thread threadB = new CounterThread(counterB);

threadA.start();

threadB.start();

}

}

Notice how the two threads, threadA and threadB, no longer reference the same counter instance. The addmethod of counterA and counterB are synchronized on their two owning instances. Calling add() on counterAwill thus not block a call to add() on counterB.

**Java Concurrency Utilities**

The synchronized mechanism was Java's first mechanism for synchronizing access to objects shared by multiple threads. The synchronized mechanism isn't very advanced though. That is why Java 5 got a whole set of [**concurrency utility classes**](http://tutorials.jenkov.com/java-util-concurrent/index.html) to help developers implement more fine grained concurrency control than what you get with synchronized.

Next: [**Java Volatile Keyword**](http://tutorials.jenkov.com/java-concurrency/volatile.html)

\_\_\_\_\_\_\_\_\_\_\_\_)))))))))\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# [Binary Semaphore vs a ReentrantLock](http://stackoverflow.com/questions/17683575/binary-semaphore-vs-a-reentrantlock)

there is no real reason ever to have a binary semaphore as everything that a binary semaphore can do can also be done by a ReentrantLock

If all you need is reentrant mutual exclusion, then yes, there is no reason to use a binary semaphore over a ReentrantLock. If for any reason you need non-ownership-release semantics then obviously semaphore is your only choice.

Also since reentrant locks also provide one lock per object, isn't it always a better idea to prefer a reentrant lock to a binary semaphore?

It depends on the need. Like previously explained, if you need a simple mutex, then don't choose a semaphore. If more than one thread (but a limited number) can enter a critical section you can do this through either thread-confinement or a semaphore.

I have checked a post here that talks about difference between a binary semaphore and a mutex but is there a thing like a mutex in Java?

ReentrantLock and synchronized are examples of mutexes in Java.

Usually you will use Reenterant lock (because you need a locking in a first place) and not a semaphore. Locks are used much often than semaphores (all semaphores, not only the binary ones).   
  
There are some small differences between Semaphore and reenterant lock. Semaphore may be released by another thread. Semaphore's javadoc states that such behavior may be useful in some specialized contexts like deadlock recovery. I never found cases where binary semaphores are better that locks. So it should be a really specialized contexts. Also binary semaphores are not reenterant. You cannot acquire a binary semaphore second time in a same thread. It will lead to a deadlock (deadlocking thread with itself!) and you may need some means of already mentioned deadlock recovery http://cache-www.coderanch.com/images/smilies/3b63d1616c5dfcf29f8a7a031aaa7cad.gif   
  
Wrong "releases" of a semaphore is not a big problem by itself. It is almost the same as wrong "releases" of the reentrant lock. And non-releasing a lock/semaphore proper number of times is much worse problem (these missed locks will be hard to spot). In most cases you should use locks in a structured manner:

[?](http://www.coderanch.com/t/615796/threads/java/reason-prefer-binary-Semaphore-Reentrant)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | lock.lock();  try {    //some code here    //use some lock.conditions to create a reason to use lock instead of synchronized  } finally {    lock.unlock();  } |

If you are accustomed to this [pattern](http://www.javaranch.com/patterns/), then it is very easy to spot missing or misplaced unlock call.

Lock () method:

---------

Acquires the lock.

\*

\* <p>Acquires the lock if it is not held by another thread and returns

\* immediately, setting the lock hold count to one.

\*

\* <p>If the current thread already holds the lock then the hold

\* count is incremented by one and the method returns immediately.

\*

\* <p>If the lock is held by another thread then the

\* current thread becomes disabled for thread scheduling

\* purposes and lies dormant until the lock has been acquired,

\* at which time the lock hold count is set to one.

Unlock :

/\*\*

\* Attempts to release this lock.

\*

\* <p>If the current thread is the holder of this lock then the hold

\* count is decremented. If the hold count is now zero then the lock

\* is released. If the current thread is not the holder of this

\* lock then {@link IllegalMonitorStateException} is thrown.

\* \* **@throws** IllegalMonitorStateException if the current thread does not

\* hold this lock \*/