# Java Synchronized Blocks




Jakob Jenkov

Last update: 2020-08-12

A *Java synchronized block* marks a method or a block of code as *synchronized*. A synchronized block in Java can only be executed a single thread at a time (depending on how you use it). Java synchronized blocks can thus be used to avoid **race conditions**. This *Java synchronized tutorial* explains how the Java *synchronized* keyword works in more detail.

## Java Synchronized Tutorial Video

If you prefer video, I have a video version of this Java synchronized tutorial here:
**Java Synchronized Tutorial**


## Java Concurrency Utilities

## 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** to help developers implement more fine grained concurrency control than what you get with `synchronized`.

# 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. Each of these synchronized blocks will be explained in more detail below.

# Synchronized Instance Methods

Here is a synchronized instance method:

```
public class MyCounter {

  private int count = 0;

  public synchronized void add(int value){
      this.count += value;
  }
}
```

Notice the use of the `synchronized` keyword in the `add()` 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 per instance 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.

This is true across all synchronized instance methods for the same object (instance). Thus, in the following example, only one thread can execute inside either of of the two synchronized methods. One thread in total per instance:

```
public class MyCounter {

  private int count = 0;
```

```
  public synchronized void add(int value){
      this.count += value;
  }
  public synchronized void subtract(int value){
      this.count -= value;
  }

}
```

# 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 MyStaticCounter{

  private static int count = 0;

  public static synchronized void add(int value){
      count += value;
  }


}
```

Also here the `synchronized` keyword tells Java that the `add()` 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.

In case a class contains more than one static synchronized method, only one thread can execute inside any of these methods at the same time. Look at this static synchronized method example:

```
public static MyStaticCounter{

  private static int count = 0;

  public static synchronized void add(int value){
    count += value;
  }

  public static synchronized void subtract(int value){
    count -= value;
  }
}
```

Only one thread can execute inside any of the two `add()` and `subtract()` methods at any given time. If Thread A is executing `add()` then Thread B cannot execute neither `add()` nor `subtract()` until Thread A has exited `add()`.

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

Synchronized blocks can also be used inside of static methods. Here are the same two examples from the previous section 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.

# Synchronized Blocks in Lambda Expressions

It is even possible to use synchronized blocks inside a **Java Lambda Expression** as well as inside anonymous classes.

Here is an example of a Java lambda expression with a synchronized block inside. Notice that the synchronized block is synchronized on the class object of the class containing the lambda expression. It could have been synchronized on another object too, if that would have made more sense (given a specific use case), but using the class object is fine for this example.

```
import java.util.function.Consumer;

public class SynchronizedExample {

  public static void main(String[] args) {

    Consumer<String> func = (String param) -> {

      synchronized(SynchronizedExample.class) {

        System.out.println(
            Thread.currentThread().getName() +
                    " step 1: " + param);

        try {
          Thread.sleep( (long) (Math.random() * 1000));
        } catch (InterruptedException e) {
          e.printStackTrace();
        }

        System.out.println(
            Thread.currentThread().getName() +
                    " step 2: " + param);
      }

    };


    Thread thread1 = new Thread(() -> {
        func.accept("Parameter");
    }, "Thread 1");

    Thread thread2 = new Thread(() -> {
        func.accept("Parameter");
    }, "Thread 2");
```

```
    thread1.start();
    thread2.start();
  }
}
```

## 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 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();
  }
}
```

Here are the two classes used in the example above, `Counter` and `CounterThread`.

```
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);
      }
   }
}
```

Two threads are created. The same `Counter` instance is passed to both of them in their constructor. The `Counter.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:

lock.

```
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 `add` method of `counterA` and `counterB` are synchronized on their two owning instances. Calling `add()` on `counterA` will thus not block a call to `add()` on `counterB`.

# Synchronized and Data Visibility

Without the use of the `synchronized` keyword (or the **Java volatile** keyword) there is no guarantee that when one thread changes the value of a variable shared with other threads (e.g. via an object all threads have access to), that the other threads can see the changed value. There are no guarantees about when a variable kept in a CPU register by one thread is "committed" to main memory, and there is no guarantee about when other threads "refresh" a variable kept in a CPU register from main memory.

The `synchronized` keyword changes that. When a thread enters a synchronized block it will refresh the values of all variables visible to the thread. When a thread exits a synchronized block all changes to variables visible to the thread will be committed to main memory. This is similar to how the **volatile keyword** works.

# Synchronized and Instruction Reordering

The Java compiler and Java Virtual Machine are allowed to reorder instructions in your code to make them execute faster, typically by enabling the reordered instructions to be executed in parallel by the CPU.

Instruction reordering could potentially cause problems in code that is executed by multiple threads at the same time. For instance, if a write to a variable happening inside of a synchronized block was reordered to happen outside of the synchronized block.

To fix this problem the Java synchronized keyword places some restrictions on reordering of instructions before, inside and after synchronized blocks. This is similar to the restrictions placed by the **volatile keyword**.

The end result is, that you can be sure that your code works correctly - that no instruction reordering is taking place that ends up making the code behave differently than what was to be expected from the code you wrote.

# What Objects to Synchronize On

As mentioned several times in this Java synchronized tutorial, a synchronized block must be synchronized on some object. You can actually choose any object to synchronize on, but it is recommended that you do not synchronize on String objects, or any primitive type wrapper objects, as the compiler might optimize those, so that you are using the same instances in different places in your code where you thought you were using different

instance. Look at this example:

```
synchronized("Hey") {
   //do something in here.
}
```

If you have more than one synchronized block that is synchronized on the literal String value "Hey", then the compiler might actually use the same String object behind the scenes. The result being, that both of these two synchronized blocks are then synchronized on the same object. That might not be the behaviour you were looking for.

The same can be true for using primitive type wrapper objects. Look at this example:

```
synchronized(Integer.valueOf(1)) {
   //do something in here.
}
```

If you call `Integer.valueOf(1)` multiple times, it might actually return the same wrapper object instance for the same input parameter values. That means, that if you are synchronizing multiple blocks on the same primitive wrapper object (e.g. use `Integer.valueOf(1)` multiple times as monitor object), then you risk that those synchronized blocks all get synchronized on the same object. That might also not be the behaviour you were looking for.

To be on the safe side, synchronize on `this` - or on a `new Object()` . Those are not cached or reused internally by the Java compiler, Java VM or Java libraries.

# Synchronized Block Limitations and Alternatives

Synchronized blocks in Java have several limitations. For instance, a synchronized block in Java only allows a single thread to enter at a time. However, what if two threads just wanted to read a shared value, and not update it? That might be safe to allow. As alternative to a synchronized block you could guard the code with a **Read / Write Lock** which as more advanced locking semantics than a synchronized block. Java actually comes with a built in **ReadWriteLock** class you can use.

What if you want to allow N threads to enter a synchronized block, and not just one? You could use a **Semaphore** to achieve that behaviour. Java actually comes with a built-in **Java Semaphore** class you can use.

Synchronized blocks do not guarantee in what order threads waiting to enter them are granted access to the synchronized block. What if you need to guarantee that threads trying to enter a synchronized block get access in the exact sequence they requested access to it? You need to implement **Fairness** yourself.

What if you just have one thread writing to a shared variable, and other threads only reading that variable? Then you might be able to just use a **volatile variable** without any synchronization around.

# Synchronized Block Performance Overhead

There is a small performance overhead associated with entering and exiting a synchronized block in Java. As Jave have evolved this performance overhead has gone down, but there is still a small price to pay.

The performance overhead of entering and exiting a synchronized block is mostly

something to worry about if you enter and exit a synchronized block lots of times within a tight loop or so.

Also, try not to have larger synchronized blocks than necessary. In other words, only synchronize the operations that are really necessary to synchronize - to avoid blocking other threads from executing operations that do not have to be synchronized. Only the absolutely necessary instructions in synchronized blocks. That should increase parallelism of your code.

## Synchronized Block Reentrance

Once a thread has entered a synchronized block the thread is said to "hold the lock" on the monitoring object the synchronized block is synchronized on. If the thread calls another method which calls back to the first method with the synchronized block inside, the thread holding the lock can reenter the synchronized block. It is not blocked just because a thread (itself) is holding the lock. Only if a differen thread is holding the lock. Look at this example:

```
public class MyClass {

  List<String> elements = new ArrayList<String>();

  public void count() {
    if(elements.size() == 0) {
        return 0;
    }
    synchronized(this) {
       elements.remove();
       return 1 + count();
    }
  }

}
```

Forget for a moment that the above way of counting the elements of a list makes no sense at all. Just focus on how inside the synchronized block inside the `count()` method calls the `count()` method recursively. Thus, the thread calling count() may eventually enter the same synchronized block multiple times. This is allowed. This is possible.

Keep in mind though, that designs where a thread enters into multiple synchronized blocks may lead to **nested monitor lockout** if you do not design your code carefully.

## Synchronized Blocks in Cluster Setups

Keep in mind that a synchronized block only blocks threads within the same Java VM from entering that code block. If you have the same Java application running on multiple Java VMs - in a cluster - then it is possible for one thread *within each Java VM* to enter that synchronized block at the same time.

If you need synchronization across all Java VMs in a cluster you will need to use other synchronization mechanisms than just a synchronized block.

Next: Java Volatile Keyword