<http://tutorials.jenkov.com/java-concurrency/thread-safety.html>

**Thread Safety and Shared Resources**

* [Local Variables](http://tutorials.jenkov.com/java-concurrency/thread-safety.html#local-variables)
* [Local Object References](http://tutorials.jenkov.com/java-concurrency/thread-safety.html#local-object-references)
* [Object Member Variables](http://tutorials.jenkov.com/java-concurrency/thread-safety.html#object-member-variables)
* [The Thread Control Escape Rule](http://tutorials.jenkov.com/java-concurrency/thread-safety.html#thread-control-escape-rule)

Code that is safe to call by multiple threads simultaneously is called *thread safe*. If a piece of code is thread safe, then it contains no [**race conditions**](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html). Race condition only occur when multiple threads update shared resources. Therefore it is important to know what resources Java threads share when executing.

**Local Variables**

Local variables are stored in each thread's own stack. That means that local variables are never shared between threads. That also means that all local primitive variables are thread safe. Here is an example of a thread safe local primitive variable:

public void someMethod(){

long threadSafeInt = 0;

threadSafeInt++;

}

**Local Object References**

Local references to objects are a bit different. The reference itself is not shared. The object referenced however, is not stored in each threads's local stack. All objects are stored in the shared heap.

If an object created locally never escapes the method it was created in, it is thread safe. In fact you can also pass it on to other methods and objects as long as none of these methods or objects make the passed object available to other threads.

Here is an example of a thread safe local object:

public void someMethod(){

LocalObject localObject = new LocalObject();

localObject.callMethod();

method2(localObject);

}

public void method2(LocalObject localObject){

localObject.setValue("value");

}

The LocalObject instance in this example is not returned from the method, nor is it passed to any other objects that are accessible from outside the someMethod() method. Each thread executing the someMethod() method will create its own LocalObject instance and assign it to the localObject reference. Therefore the use of the LocalObject here is thread safe.

In fact, the whole method someMethod() is thread safe. Even if the LocalObject instance is passed as parameter to other methods in the same class, or in other classes, the use of it is thread safe.

The only exception is of course, if one of the methods called with the LocalObject as parameter, stores the LocalObject instance in a way that allows access to it from other threads.

**Object Member Variables**

Object member variables (fields) are stored on the heap along with the object. Therefore, if two threads call a method on the same object instance and this method updates object member variables, the method is not thread safe. Here is an example of a method that is not thread safe:

public class NotThreadSafe{

StringBuilder builder = new StringBuilder();

public add(String text){

this.builder.append(text);

}

}

If two threads call the add() method simultaneously **on the same NotThreadSafe instance** then it leads to race conditions. For instance:

NotThreadSafe sharedInstance = new NotThreadSafe();

new Thread(new MyRunnable(sharedInstance)).start();

new Thread(new MyRunnable(sharedInstance)).start();

public class MyRunnable implements Runnable{

NotThreadSafe instance = null;

public MyRunnable(NotThreadSafe instance){

this.instance = instance;

}

public void run(){

this.instance.add("some text");

}

}

Notice how the two MyRunnable instances share the same NotThreadSafe instance. Therefore, when they call the add() method on the NotThreadSafe instance it leads to race condition.

However, if two threads call the add() method simultaneously **on different instances** then it does not lead to race condition. Here is the example from before, but slightly modified:

new Thread(new MyRunnable(new NotThreadSafe())).start();

new Thread(new MyRunnable(new NotThreadSafe())).start();

Now the two threads have each their own instance of NotThreadSafe so their calls to the add method doesn't interfere with each other. The code does not have race condition anymore. So, even if an object is not thread safe it can still be used in a way that doesn't lead to race condition.

**The Thread Control Escape Rule**

When trying to determine if your code's access of a certain resource is thread safe you can use the thread control escape rule:

If a resource is created, used and disposed within

the control of the same thread,

and never escapes the control of this thread,

the use of that resource is thread safe.

Resources can be any shared resource like an object, array, file, database connection, socket etc. In Java you do not always explicitly dispose objects, so "disposed" means losing or null'ing the reference to the object.

Even if the use of an object is thread safe, if that object points to a shared resource like a file or database, your application as a whole may not be thread safe. For instance, if thread 1 and thread 2 each create their own database connections, connection 1 and connection 2, the use of each connection itself is thread safe. But the use of the database the connections point to may not be thread safe. For example, if both threads execute code like this:

check if record X exists

if not, insert record X

If two threads execute this simultaneously, and the record X they are checking for happens to be the same record, there is a risk that both of the threads end up inserting it. This is how:

Thread 1 checks if record X exists. Result = no

Thread 2 checks if record X exists. Result = no

Thread 1 inserts record X

Thread 2 inserts record X

This could also happen with threads operating on files or other shared resources. Therefore it is important to distinguish between whether an object controlled by a thread **is** the resource, or if it merely **references** the resource (like a database connection does).

<http://tutorials.jenkov.com/java-concurrency/thread-safety-and-immutability.html>

**Thread Safety and Immutability**

* [The Reference is not Thread Safe!](http://tutorials.jenkov.com/java-concurrency/thread-safety-and-immutability.html#reference)

[**Race conditions**](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html) occur only if multiple threads are accessing the same resource, **and** one or more of the threads **write** to the resource. If multiple threads read the same resource [**race conditions**](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html) do not occur.

We can make sure that objects shared between threads are never updated by any of the threads by making the shared objects immutable, and thereby thread safe. Here is an example:

public class ImmutableValue{

private int value = 0;

public ImmutableValue(int value){

this.value = value;

}

public int getValue(){

return this.value;

}

}

Notice how the value for the ImmutableValue instance is passed in the constructor. Notice also how there is no setter method. Once an ImmutableValue instance is created you cannot change its value. It is immutable. You can read it however, using the getValue() method.

If you need to perform operations on the ImmutableValue instance you can do so by returning a new instance with the value resulting from the operation. Here is an example of an add operation:

public class ImmutableValue{

private int value = 0;

public ImmutableValue(int value){

this.value = value;

}

public int getValue(){

return this.value;

}

**public ImmutableValue add(int valueToAdd){**

**return new ImmutableValue(this.value + valueToAdd);**

**}**

}

Notice how the add() method returns a new ImmutableValue instance with the result of the add operation, rather than adding the value to itself.

**The Reference is not Thread Safe!**

It is important to remember, that even if an object is immutable and thereby thread safe, the reference to this object may not be thread safe. Look at this example:

public class Calculator{

private ImmutableValue currentValue = null;

public ImmutableValue getValue(){

return currentValue;

}

public void setValue(ImmutableValue newValue){

this.currentValue = newValue;

}

public void add(int newValue){

this.currentValue = this.currentValue.add(newValue);

}

}

The Calculator class holds a reference to an ImmutableValue instance. Notice how it is possible to change that reference through both the setValue() and add() methods. Therefore, even if the Calculator class uses an immutable object internally, it is not itself immutable, and therefore not thread safe. In other words: The ImmutableValue class is thread safe, but the **use of it** is not. This is something to keep in mind when trying to achieve thread safety through immutability.

To make the Calculator class thread safe you could have declared the getValue(), setValue(), and add() methods synchronized. That would have done the trick.

# [Threadsafe vs Synchronized](https://stackoverflow.com/questions/21792030/threadsafe-vs-synchronized)

<https://stackoverflow.com/questions/21792030/threadsafe-vs-synchronized>

Q: I'm new to java. I'm little bit confused between Threadsafe and synchronized. Thread safe means that a method or class instance can be used by multiple threads at the same time without any problems occurring. Where as Synchronized means only one thread can operate at single time.

A:

The definition of thread safety given in [Java Concurrency in Practice](http://amzn.to/1jyE5Kx) is:

A class is thread-safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

For example, a java.text.SimpleDateFormat object has internal mutable state that is modified when a method that parses or formats is called. If multiple threads call the methods of the same dateformat object, there is a chance a thread can modify the state needed by the other threads, with the result that the results obtained by some of the threads may be in error. The possibility of having internal state get corrupted causing bad output makes this class not threadsafe.

There are multiple ways of handling this problem. You can have every place in your application that needs a SimpleDateFormat object instantiate a new one every time it needs one, you can make a ThreadLocal holding a SimpleDateFormat object so that each thread of your program can access its own copy (so each thread only has to create one), you can use an alternative to SimpleDateFormat that doesn't keep state, or you can do locking using synchronized so that only one thread at a time can access the dateFormat object.

Locking is not necessarily the best approach, avoiding shared mutable state is best whenever possible. That's why in Java 8 they introduced a date formatter that doesn't keep mutable state.

The synchronized keyword is one way of restricting access to a method or block of code so that otherwise thread-unsafe data doesn't get corrupted. This keyword protects the method or block by requiring that a thread has to acquire exclusive access to a certain lock (the object instance, if synchronized is on an instance method, or the class instance, if synchronized is on a static method, or the specified lock if using a synchronized block) before it can enter the method or block, while providing memory visibility so that threads don't see stale data.

# Java synchronized keyword

<https://howtodoinjava.com/java/keywords/java-synchronized/>

**Java synchronized keyword** marks a block or method a critical section. A critical section is where one and only one thread is executing at a time, and the thread holds the lock for the synchronized section.

**synchronized** keyword helps in writing [concurrent](https://howtodoinjava.com/java-concurrency-tutorial/) parts of the applications, to protect shared resources within this block.

The synchronized keyword can be use with –

* a code block
* a method

## 1. Java synchronized block

#### 1.1. Syntax

The general syntax for writing a synchronized block is as follows. Here **lockObject** is a reference to an object whose lock associates with the monitor that the synchronized statements represent.

|  |
| --- |
| Syntax |
| synchronized( lockObject )  {     // synchronized statements  } |

#### 1.2. Internal working

When a thread wants to execute synchronized statements inside the synchronized block, it MUST acquire the lock on lockObject‘s monitor. At a time, only one thread can acquire the monitor of a lock object. So all other threads must wait till this thread, currently acquired the lock, finish it’s execution.

In this way, synchronized keyword guarantees that only one thread will be executing the synchronized block statements at a time, and thus prevent multiple threads from corrupting the shared data inside the block.

Keep in mind that if a thread is put on sleep (using sleep() method) then it does not release the lock. At this sleeping time, no thread will be executing the synchronized block statements.

Java synchronization will throw **NullPointerException** if lock object used in 'synchronized (lock)' is null.

#### 1.3. Java synchronized block example

Java program to demonstrate the usage of synchronized block. In given example, we have a MathClass with a method printNumbers(). This method will print the numbers starting from 1 to the argument number N.

Notice that the code in printNumbers() method is inside synchronized block.

|  |
| --- |
| MathClass.java |
| public class MathClass  {      void printNumbers(int n) throws InterruptedException      {          synchronized (this)          {              for (int i = 1; i <= n; i++)              {                  System.out.println(Thread.currentThread().getName() + " :: "+  i);                  Thread.sleep(500);              }          }      }  } |

I have created two threads which start executing the printNumbers() method exactly at same time. Due to block being synchronized, only one thread is allowed access and other thread has to wait until first thread is finished.

|  |
| --- |
| Main.java |
| public class Main  {      public static void main(String args[])      {          final MathClass mathClass = new MathClass();            //first thread          Runnable r = new Runnable()          {              public void run()              {                  try {                      mathClass.printNumbers(3);                  } catch (InterruptedException e) {                      e.printStackTrace();                  }              }          };            new Thread(r, "ONE").start();          new Thread(r, "TWO").start();      }  } |

Program output.

|  |
| --- |
| Console |
| ONE :: 1  ONE :: 2  ONE :: 3    TWO :: 1  TWO :: 2  TWO :: 3 |

## 2. Java synchronized method

#### 2.1. Syntax

The general syntax for writing a synchronized method is as follows. Here **lockObject** is a reference to an object whose lock associates with the monitor that the synchronized statements represent.

|  |
| --- |
| Syntax |
| <access modifier> synchronized method( parameters )  {      // synchronized code  } |

#### 2.2. Internal working

Similar to synchronized block, a thread MUST acquire the lock on the associated monitor object with synchronized method. In case of synchronized method, the lock object is –

* **‘.class’ object** – if the method is static.
* **‘this’ object** – if the method is not static. ‘this’ refer to reference to current object in which synchronized method is invoked.

Read More : [Object level lock vs Class level lock in Java](https://howtodoinjava.com/java/multi-threading/object-vs-class-level-locking/)

Java synchronized keyword is **re-entrant** in nature it means if a synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.

#### 2.3. Java synchronized method example

Similar to synchronized block example, we can apply synchronized keyword at printNumber() method and it will make the method as synchronized. Now if we again run the example, we will get the similar output.

|  |
| --- |
| MathClass.java |
| public class MathClass  {      synchronized void printNumbers(int n) throws InterruptedException      {          for (int i = 1; i <= n; i++)          {              System.out.println(Thread.currentThread().getName() + " :: "+  i);              Thread.sleep(500);          }      }  } |

Program output.

|  |
| --- |
| Console |
| ONE :: 1  ONE :: 2  ONE :: 3    TWO :: 1  TWO :: 2  TWO :: 3 |

# Object level lock vs Class level lock in Java

<https://howtodoinjava.com/java/multi-threading/object-vs-class-level-locking/>

In Java, a **synchronized** block of code can only be executed by one thread at a time. Also, java supports multiple threads to be executed concurrently. This may cause two or more threads to access the same fields or objects at same time.

Synchronization is the process which keeps all concurrent threads in execution to be in sync. Synchronization avoids memory consistence errors caused due to inconsistent view of shared memory. When a method is declared as **synchronized**; the thread holds the monitor or [**lock**](https://howtodoinjava.com/java/multi-threading/how-to-use-locks-in-java-java-util-concurrent-locks-lock-tutorial-and-example/)**object** for that method’s object. If another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor.

Please note that we can use synchronized keyword in the class on defined methods or blocks. synchronized keyword can not be used with variables or attributes in class definition.

## 1. Object level lock in Java

**Object level lock** is mechanism when we want to synchronize a **non-static method** or **non-static code block** such that only one thread will be able to execute the code block on given instance of the class. This should always be done **to make instance level data thread safe**.

Object level locking can be done as below :

|  |
| --- |
| Various ways for object level locking |
| public class DemoClass  {      public synchronized void demoMethod(){}  }    or    public class DemoClass  {      public void demoMethod(){          synchronized (this)          {              //other thread safe code          }      }  }    or    public class DemoClass  {      private final Object lock = new Object();      public void demoMethod(){          synchronized (lock)          {              //other thread safe code          }      }  } |

## 2. Class level lock in Java

**Class level lock** prevents multiple threads to enter in synchronized block in any of all available instances of the class on runtime. This means if in runtime there are 100 instances of DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads.

Class level locking should always be done **to make static data thread safe**. As we know that [**static**](https://howtodoinjava.com/java/keywords/java-static-keyword/) keyword associate data of methods to class level, so use locking at static fields or methods to make it on class level.

|  |
| --- |
| Various ways for class level locking |
| public class DemoClass  {      //Method is static      public synchronized static void demoMethod(){        }  }    or    public class DemoClass  {      public void demoMethod()      {          //Acquire lock on .class reference          synchronized (DemoClass.class)          {              //other thread safe code          }      }  }    or    public class DemoClass  {      private final static Object lock = new Object();        public void demoMethod()      {          //Lock object is static          synchronized (lock)          {              //other thread safe code          }      }  } |

## 3. Object level lock vs class level lock – Important notes

1. Synchronization in Java guarantees that no two threads can execute a synchronized method, which requires same lock, simultaneously or concurrently.
2. synchronized keyword can be used only with methods and code blocks. These methods or blocks can be static or non-static both.
3. When ever a thread enters into Java synchronized method or block it acquires a lock and whenever it leaves synchronized method or block it releases the lock. Lock is released even if thread leaves synchronized method after completion or due to any Error or Exception.
4. Java synchronized keyword is **re-entrant** in nature it means if a synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.
5. Java synchronization will throw [NullPointerException](https://howtodoinjava.com/java/exception-handling/how-to-effectively-handle-nullpointerexception-in-java/) if object used in synchronized block is null. For example, in above code sample if lock is initialized as null, the “synchronized (lock)” will throw NullPointerException.
6. Synchronized methods in Java put a performance cost on your application. So use synchronization when it is absolutely required. Also, consider using synchronized code blocks for synchronizing only critical section of your code.
7. It’s possible that both static synchronized and non static synchronized method can run simultaneously or concurrently because they lock on different object.
8. According to the Java language specification you can not use synchronized keyword with constructor. It is illegal and result in compilation error.
9. Do not synchronize on non final field on synchronized block in Java. because reference of non final field may change any time and then different thread might synchronizing on different objects i.e. no synchronization at all.
10. Do not use String literals because they might be referenced else where in the application and can cause deadlock. String objects created with new keyword can be used safely. But as a best practice, create a new **private** scoped Object instance OR lock on the shared variable itself which we want to protect.