# Java Concurrency – Thread Safety?

Defining **thread safety** is surprisingly tricky.

1. Thread-safe code is code that will work even if many Threads are executing it simultaneously.
2. A piece of code is thread-safe if it only manipulates shared data structures in a manner that guarantees safe execution by multiple threads at the same time.

And there are more similar definitions.

Don’t you think that definitions like above actually does not communicate anything meaningful and even add some more confusion. Though these definitions can’t be ruled out just like that, because they are not wrong. But the fact is **they do not provide any practical help or perspective**. How do we make a **difference between a thread-safe class and an unsafe one**? **What do we even mean by “safe”**?

## What is Correctness in thread safety?

At the heart of any reasonable definition of thread safety is the concept of correctness. So, before understanding the thread-safety we should understand first, this “correctness“.

Correctness means that a class conforms to its specification.

You will agree that a good class specification will have all information about a class’s state at any given time and it’s post condition if some operation is performed on it. Since we often don’t write adequate specifications for our classes, how can we possibly know they are correct? We can’t, but that doesn’t stop us from using them anyway once we’ve convinced ourselves that “the code works”. This **“code confidence**” is about as close as many of us get to correctness.

Having optimistically defined “correctness” as something that can be recognized, we can now define thread safety in a somewhat less circular way: **a class is thread-safe when it continues to behave correctly when accessed from multiple threads**.

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.

If the loose use of “correctness” here bothers you, you may prefer to **think of a thread-safe class as one that is no more broken in a concurrent environment than in a single-threaded environment**. Thread-safe classes encapsulate any needed synchronization so that clients need not provide their own.

**Concurrency vs. Parallelism**

## Concurrency

Concurrency is essentially applicable when we talk about minimum two tasks or more. When an application is capable of executing two tasks virtually at same time, we call it concurrent application. Though here tasks run looks like simultaneously, but essentially they MAY not. They take advantage of **CPU time-slicing** feature of operating system where each task run part of its task and then go to waiting state. When first task is in waiting state, CPU is assigned to second task to complete it’s part of task.

Operating system based on priority of tasks, thus, assigns CPU and other computing resources e.g. memory; turn by turn to all tasks and give them chance to complete. To end user, it seems that all tasks are running in parallel. This is called concurrency.

## Parallelism

Parallelism does not require two tasks to exist. It literally physically run parts of tasks OR multiple tasks, at the same time using multi-core infrastructure of CPU, by assigning one core to each task or sub-task.

Parallelism requires hardware with multiple processing units, essentially. In single core CPU, you may get concurrency but NOT parallelism.

## Differences between concurrency vs. parallelism

Now let’s list down remarkable differences between concurrency and parallelism.

Concurrency is when two tasks can start, run, and complete in overlapping time periods. Parallelism is when tasks literally run at the same time, eg. on a multi-core processor.

Concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations.

Concurrency is about **dealing with lots of things** at once. Parallelism is about **doing lots of things** at once.

An application can be concurrent – but not parallel, which means that it processes more than one task at the same time, but no two tasks are executing at same time instant.

An application can be parallel – but not concurrent, which means that it processes multiple sub-tasks of a task in multi-core CPU at same time.

An application can be neither parallel – nor concurrent, which means that it processes all tasks one at a time, sequentially.

An application can be both parallel – and concurrent, which means that it processes multiple tasks concurrently in multi-core CPU at same time .

# Object level lock vs Class level lock in Java

By Lokesh Gupta | Filed Under: [Java Concurrency](https://howtodoinjava.com/java/multi-threading/)

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/basics/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. [Thanks to Anu to point this out in comments.]