**Java Concurrency Tutorial**

In simple words, [**concurrency**](https://en.wikipedia.org/wiki/Concurrency_%28computer_science%29) is the ability to run several programs or several parts of a program in parallel. Concurrency enable a program to achieve high performance and throughput by utilizing the untapped capabilities of underlying operating system and machine hardware. e.g. modern computers has several CPU’s or several cores within one CPU, program can utilize all cores for some part of processing; thus completing task much before in time in comparison to sequential processing.

The backbone of **java concurrency** are threads. A thread is a lightweight process which has its own call stack, but can access shared data of other threads in the same process. A Java application runs by default in one process. Within a Java application you can work with many threads to achieve parallel processing or concurrency.

**What makes java application concurrent?**

The very first class, you will need to make a java class concurrent, is [java.lang.Thread](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html" \t "_blank) class. This class is the basis of all concurrency concepts in java. Then you have [java.lang.Runnable](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html" \t "_blank) interface to abstract the thread behavior out of thread class.

Other classes you will need to build advance applications can be found at [java.util.concurrent](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html" \t "_blank) package added in Java 1.5.

**Java Concurrency – Thread Safety?**

Defining **thread safety** is surprisingly tricky.

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. How do we make a **difference between a thread-safe class and an unsafe one**? **What do we even mean by “safe”**?

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.

You may also 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.

The transient state for a particular computation exists solely in local variables that are stored on the thread’s stack and are accessible only to the executing thread. ne thread accessing a class cannot influence the result of another thread accessing the same class; because the two threads do not share state, it is as if they were accessing different instances. Since the actions of a thread accessing a stateless object cannot affect the correctness of operations in other threads, stateless objects are thread-safe.

# Object level lock vs Class level lock in Java

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.

**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**.

## 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.

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

        }

    }

}

**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 cannot use synchronized keyword with constructor. It is illegal and result in compilation error.
9. Do not synchronize on nonfinal 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.