<https://howtodoinjava.com/java/multi-threading/concurrency-vs-parallelism/>

<https://howtodoinjava.com/java-collections/>

# ***Concurrency vs. Parallelism***

**Concurrency** means multiple tasks which start, run, and complete in overlapping time periods, in no specific order. **Parallelism** is when multiple tasks OR several part of a unique task literally run at the same time, e.g. on a multi-core processor. Remember that Concurrency and parallelism are NOT the same thing.

## **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.***

***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 .*

# ***Java Compare and Swap Example – CAS Algorithm***

One of the best additions in java 5 was Atomic operations supported in classes such as AtomicInteger, AtomicLong etc. These classes help you in minimizing the need of complex (un-necessary) [**multi-threading**](https://howtodoinjava.com/category/java/multi-threading/) code for some basic operations such as increment or decrement a value which is shared among multiple threads. These classes internally rely on an algorithm named **CAS (compare and swap)**.

## 1. ***Optimistic and Pessimistic Locking***

Traditional locking mechanisms, e.g. **using synchronized keyword in java, is said to be pessimistic technique** of locking or multi-threading. It asks you to first guarantee that no other thread will interfere in between certain operation (i.e. lock the object), and then only allow you access to any instance/method.

Though above approach is safe and it does work, but it **put a significant penalty on your application in terms of performance**. Reason is simple that waiting threads can not do anything unless they also get a chance and perform the guarded operation.

There exist one more approach which is more efficient in performance, and it **optimistic** in nature. In this approach, you proceed with an update, **being hopeful that you can complete it without interference**. This approach relies on collision detection to determine if there has been interference from other parties during the update, in which case the operation fails and can be retried (or not).

**Compare and Swap** is a good example of such optimistic approach, which we are going to discuss next.

## ***2. Compare and Swap Algorithm***

This algorithm compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail. The result of the operation must indicate whether it performed the substitution; this can be done either with a simple Boolean response (this variant is often called compare-and-set), or by returning the value read from the memory location (not the value written to it).

There are 3 parameters for a CAS operation:

1. A memory location V where value has to be replaced
2. Old value A which was read by thread last time
3. New value B which should be written over V

CAS says “I think V should have the value A; if it does, put B there, otherwise don’t change it but tell me I was wrong.” CAS is an optimistic technique—it proceeds with the update in the hope of success, and can detect failure if another thread has updated the variable since it was last examined.

## 3. ***Java Compare and Swap Example***

Let’s understand the whole process with an example. Assume V is a memory location where value “10” is stored. There are multiple threads who want to increment this value and use the incremented value for other operations, a very practical scenario. Let’s break the whole CAS operation in steps:

**1) Thread 1 and 2 want to increment it, they both read the value and increment it to 11.**

V = 10, A = 0, B = 0

**2) Now thread 1 comes first and compare V with it’s last read value:**

V = 10, A = 10, B = 11

if     A = V

   V = B

else

   operation failed

   return V

Clearly the value of V will be overwritten as 11, i.e. operation was successful.

**3) Thread 2 comes and try the same operation as thread 1**

V = 11, A = 10, B = 11

if     A = V

   V = B

else

   operation failed

   return V

**4) In this case, V is not equal to A, so value is not replaced and current value of V i.e. 11 is returned. Now thread 2, again retry this operation with values:**

V = 11, A = 11, B = 12

And this time, condition is met and incremented value 12 is returned to thread 2.

In summary, when multiple threads attempt to update the same variable simultaneously using CAS, one wins and updates the variable’s value, and the rest lose. But the losers are not punished by suspension of thread. They are free to retry the operation or simply do nothing.

# ***Object level lock vs Class level lock in Java***

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**.

|  |
| --- |
| 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.]

## ***Java sleep() vs wait() – Summary***

#### 3.1. Method called on

* wait() – Call on an object; current thread must synchronize on the lock object.
* sleep() – Call on a Thread; always currently executing thread.

#### 3.2. Synchronized

* wait() – when synchronized multiple threads access same Object one by one.
* sleep() – when synchronized multiple threads wait for sleep over of sleeping thread.

#### 3.3. Lock duration

* wait() – release the lock for other objects to have chance to execute.
* sleep() – keep lock for at least t times if timeout specified or somebody interrupt.

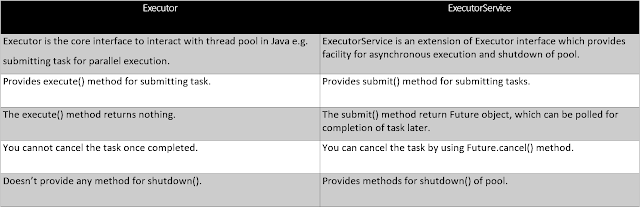
#### 3.4. wake up condition

* wait() – until call notify(), notifyAll() from object
* sleep() – until at least time expire or call interrupt().

#### 3.5. Usage

* sleep() – for time-synchronization
* wait() – for multi-thread-synchronization.
* **What are Executor and ExecutorService? What are the differences between these interfaces?**

One of the key difference between Executor and ExecutorService interface is that former is a parent interface while ExecutorService extends Executor i.e. it's a sub-interface of Executor.  
  
2) Another important difference between ExecutorService and Executor is that Executor defines execute() method which accepts an object of the Runnable interface, while submit() method can accept objects of both [Runnable](http://www.java67.com/2016/01/7-differences-between-extends-thread-vs-implements-Runnable-java.html) and [Callable](http://javarevisited.blogspot.com/2016/08/useful-difference-between-callable-and-Runnable-in-Java.html) interfaces.  
  
3) The third difference between Executor and ExecutorService interface is that execute() method doesn't return any result, its return type is void but submit() method returns the result of computation via a [Future](http://javarevisited.blogspot.com/2015/06/how-to-use-callable-and-future-in-java.html) object. This is also the key difference between submit() and execute() method, which is one of the frequently asked [Java concurrency interview questions](http://javarevisited.blogspot.sg/2014/07/top-50-java-multithreading-interview-questions-answers.html).  
  
4) The fourth difference between ExecutorService and Executor interface is that apart from allowing a client to submit a task, ExecutorService also provides methods to control the thread pool e.g. terminate the thread pool by calling the shutDown() method. You should also read ["Java Concurrency in Practice"](http://www.amazon.com/dp/0321349601/?tag=javamysqlanta-20) to learn more about the graceful shutdown of a thread-pool and how to handle pending tasks.

[](https://4.bp.blogspot.com/--CAPMy-yet0/WLYI_SQvMuI/AAAAAAAAIC0/VyDHWmgCwuw38ZTyF2o_MiiCFwhbt78_wCLcB/s1600/Difference%2Bbetween%2BExecutor%2Band%2BExecutorService%2Bin%2BJava.png)

* **What are the available implementations of ExecutorService in the standard library?**

The ExecutorService interface has three standard implementations:

**ThreadPoolExecutor** — for executing tasks using a pool of threads. Once a thread is finished executing the task, it goes back into the pool. If all threads in the pool are busy, then the task has to wait for its turn.

***ScheduledThreadPoolExecutor*** allows to schedule task execution instead of running it immediately when a thread is available. It can also schedule tasks with fixed rate or fixed delay.

***ForkJoinPool*** is a special ExecutorService for dealing with recursive algorithms tasks. If you use a regular ThreadPoolExecutor ***for a recursive algorithm***, you will quickly find all your threads are busy waiting for the lower levels of recursion to finish. *The ForkJoinPool implements the so-called work-stealing algorithm that allows it to use available threads more efficiently.*

* **What special guarantees does the JMM hold for final fields of a class?**

JVM basically guarantees that final fields of a class will be initialized before any thread gets hold of the object. Without this guarantee, a reference to an object may be published, i.e. become visible, to another thread before all the fields of this object are initialized, due to reorderings or other optimizations. This could cause racy access to these fields.

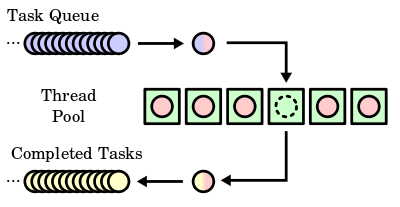
This is why, when creating an immutable object, you should always make all its fields final, even if they are not accessible via getter methods.

# ***Java Thread Pool – ThreadPoolExecutor***

## 1. ***How thread pool works in java***

A **thread pool is a collection of pre-initialized threads**. Generally the size of collection is fixed, but it is not mandatory. It facilitates the execution of N number of tasks using same threads. If thread are more tasks than threads, then tasks need to wait in a queue like structure ([FIFO – First in first out](https://en.wikipedia.org/wiki/FIFO_and_LIFO_accounting#FIFO)).

When any thread completes it’s execution, it can pickup a new task from queue and execute it. When all tasks are completed the threads remain active and wait for more tasks in thread pool.

Thread Pool

A watcher keep watching queue (usually [BlockingQueue](https://howtodoinjava.com/java/multi-threading/how-to-use-blockingqueue-and-threadpoolexecutor-in-java/)) for any new tasks. As soon as tasks come, threads again start picking up tasks and execute them.

## 2. ***ThreadPoolExecutor***

Since Java 5, the Java concurrency API provides a mechanism **Executor framework**. This is around the Executor interface, its sub-interface [ExecutorService](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html), and the ThreadPoolExecutor class that implements both interfaces.

ThreadPoolExecutor separates the task creation and its execution. With ThreadPoolExecutor, you only have to implement the Runnable objects and send them to the executor. It is responsible for their execution, instantiation, and running with necessary threads.

It goes beyond that and improves performance using a pool of threads. When you send a task to the executor, it tries to use a pooled thread for the execution of this task, to avoid continuous spawning of threads.

## 3. ***How to create ThreadPoolExecutor***

We can create following 5 types of thread pool executors with pre-built methods in java.util.concurrent.Executors interface.

1. **Fixed thread pool executor** – Creates a thread pool that reuses a fixed number of threads to execute any number of tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. It is best fit for most off the real-life usecases.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(10); |

1. **Cached thread pool executor** – Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available***. DO NOT use this thread pool if tasks are long running. It can bring down the system if number of threads goes beyond what system can handle.***

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newCachedThreadPool(); |

1. **Scheduled thread pool executor** – Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newScheduledThreadPool(10); |

1. **Single thread pool executor** – Creates single thread to execute all tasks. ***Ute it when you have only one task to execute.***

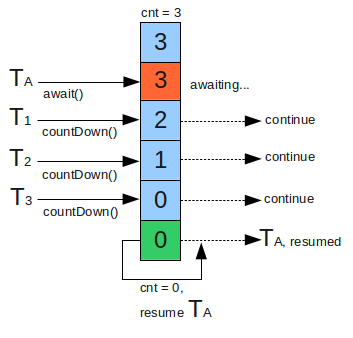
|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newSingleThreadExecutor(); |

1. **Work stealing thread pool executor** – Creates a thread pool that maintains enough threads to support the given parallelism level. Here parallelism level means the maximum number of threads which will be used to execute a given task, at single point of time, in multi-processor machines.

|  |
| --- |
| ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newWorkStealingPool(4); |

## ***What is CountDownLatch?***

CountDownLatch works by having a counter initialized with number of threads, which is decremented each time a thread complete its execution. When count reaches to zero, it means all threads have completed their execution, and thread waiting on latch resume the execution.

CountDownLatch Concept

Pseudo code for CountDownLatch can be written like this:

//Main thread start

//Create CountDownLatch for N threads

//Create and start N threads

//Main thread wait on latch

//N threads completes there tasks are returns

//Main thread resume execution

## **How CountDownLatch works?**

CountDownLatch.java class defines one constructor inside:

|  |
| --- |
| //Constructs a CountDownLatch initialized with the given count.  public CountDownLatch(int count) {...} |

This **count is essentially the number of threads**, for which latch should wait. This value can be set only once, and CountDownLatch provides **no other mechanism to reset this count**.

The first interaction with CountDownLatch is with main thread which is going to wait for other threads. This main thread must call, **CountDownLatch.await()** method immediately after starting other threads. The execution will stop on await() method till the time, other threads complete their execution.

Other N threads must have reference of latch object, because they will need to notify the CountDownLatch object that they have completed their task. This notification is done by method : **CountDownLatch.countDown()**; Each invocation of method decreases the initial count set in constructor, by 1. So, when all N threads have call this method, count reaches to zero, and main thread is allowed to resume its execution past await() method.

## **Possible usages in real time applications**

Let’s try to identify some possible usage of CountDownLatch in real time java applications. I am listing, as much i can recall. If you have any other possible usage, please leave a comment. It will help others.

1. **Achieving Maximum Parallelism** : Sometimes we want to start a number of threads at the same time to achieve maximum parallelism. For example, we want to test a class for being singleton. This can be done easily if we create a CountDownLatch with initial count 1, and make wait all threads to wait of latch. ***A single call to countDown() method will resume execution for all waiting threads in same time***.
2. **Wait N threads to completes before start execution**: For example an application start-up class want to ensure that all N external systems are Up and running before handling the user requests.
3. **Deadlock detection**: A very handy use case in which you can use N threads to access a shared resource with different number of threads in each test phase, and try to create a deadlock.