**CountDownLatch**

CountDownLatch works in latch principle that the main thread will wait until the gate is open ie. Other threads have completed the tasks. Any thread, calling **CountDownLatch.await()** will wait until count reaches zero or it's interrupted by another thread. All other threads are required to count down by calling **CountDownLatch.countDown()** once they are completed or ready.

**One of the disadvantages/advantages of CountDownLatch is that it's not reusable**: once count reaches zero you cannot use CountDownLatch any more.

Example: Consider an IT world scenario where the manager divided modules between development teams (A and B) and wants to assign them to the QA team for testing only when both teams complete their task.

Here manager thread works as the main thread and the development team works as the worker thread. The manager thread waits for the development team's thread to complete their task. Once developer teams complete their tasks, they will inform the manager thread, and then the manager thread assign modules to the QA team.

**Why use CountDownLatch (rather than wait/notify, Condition etc)?**

**The CountDownLatch protects you against the case of a thread missing a signal** which can occur if you use these other mechanisms for coordinating jobs.

Problem🡺 **A teacher gives a problem to 5 students to compute. Once all the students have completed the computations, finally teacher has to disclose the result of all students.** Code is given below.

Problem🡺 **Bank account will be opened after the validations of Aadhar, Pan and Passport**

**A precise way to writing CountDownLatch using Java 8 Lambda**

**public class** TestLatch {  
  
 **public static void** task(CountDownLatch latch, String name, **int** time) {  
 **try** {  
 Thread.*currentThread*().setName(name);  
 System.***out***.println(**"Executing Task ...."**+Thread.*currentThread*().getName());  
 TimeUnit.***SECONDS***.sleep(time);  
 System.***out***.println(**"Task completed ..."**);  
 } **catch**(InterruptedException ie) {  
 ie.printStackTrace();  
 } **finally** {  
 latch.countDown();  
 }  
 }  
  
 **public static void** main(String[] args) **throws** InterruptedException {  
 CountDownLatch latch = **new** CountDownLatch(3);  
  
 Runnable r1 = () -> *task*(latch, **"Aadhar"**,7);  
 Runnable r2 = () -> *task*(latch, **"Pan"**, 5);  
 Runnable r3 = () -> *task*(latch, **"Passport"**, 3);Thread t1 = **new** Thread(r1);  
 Thread t2 = **new** Thread(r1);  
 Thread t3 = **new** Thread(r1);  
  
 t1.start();  
 t2.start();  
 t3.start();  
  
 latch.await();  
 System.***out***.println(**"All validations completed "**);  
 }  
}

The above can also be achieved using CompletableFuture.allOf().

CompletableFuture.*allOf*(cf1,cf2,cf3).whenComplete((result,error) -> {  
 System.***out***.println(result);  
}).join();

**Difference between CountDownLatch and CyclicBarrier**

* In **CountDownLatch, threads waits for other threads to complete their execution**. In **CyclicBarrier, worker threads wait for each other to complete their execution**.
* **In short, CyclicBarrier maintains a count of threads whereas CountDownLatch maintains a count of tasks.**
* **Latches are for waiting for events**; **barriers are for waiting for other threads**.
* When using a CyclicBarrier, the assumption is that you specify the number of waiting threads that trigger the barrier. If you specify 5, you must have at least 5 threads to call await().
* When using a CountDownLatch, you specify the number of calls to countDown() that will result in all waiting threads being released. This means that you can use a CountDownLatch with only a single thread.

**CountDownLatch**: A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

**CyclicBarrier**: A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.