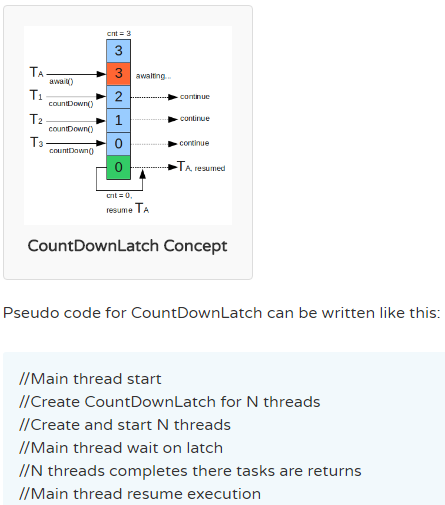
## **What is CountDownLatch?**

CountDownLatch was **introduced with JDK 1.5 along with other concurrent utilities like CyclicBarrier, Semaphore,**[**ConcurrentHashMap**](https://howtodoinjava.com/core-java/collections/best-practices-for-using-concurrenthashmap/)**and**[**BlockingQueue**](https://howtodoinjava.com/java-5/how-to-use-blockingqueue-and-threadpoolexecutor-in-java/) in java.util.concurrent package. This class **enables a java thread to wait until other set of threads completes** their tasks. e.g. Application’s main thread want to wait, till other service threads which are responsible for starting framework services have completed started all services.

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.



[Executor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html) just executes stuff you give it.

[ExecutorService](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html) adds startup, shutdown, and the ability to wait for and look at the status of jobs you've submitted for execution on top of Executor (which it extends).

## **=====A CountDownLatch's latch.await() method vs Thread.join()=====**

## The **CountDownLatch** class allows us to coordinate the starting and stopping of threads. Typical uses are as follows: We can make several threads start at the same time; We can wait for several threads to finish (whereas, for example, the Thread.**join**() method only lets you wait for a single thread).

## **Example application using CountDownLatch**

In this example, I have simulated an application startup class which starts N threads that will check for external systems and report back to latch, on which startup class is waiting. As soon as all services are verified and checked, startup proceeds.

**BaseHealthChecker.java** : This class is a Runnable and parent for all specific external service health checkers. This remove the code duplicacy and central control over latch.

|  |
| --- |
| public abstract class BaseHealthChecker implements Runnable {        private CountDownLatch \_latch;      private String \_serviceName;      private boolean \_serviceUp;        //Get latch object in constructor so that after completing the task, thread can countDown() the latch      public BaseHealthChecker(String serviceName, CountDownLatch latch)      {          super();          this.\_latch = latch;          this.\_serviceName = serviceName;          this.\_serviceUp = false;      }        @Override      public void run() {          try {              verifyService();              \_serviceUp = true;          } catch (Throwable t) {              t.printStackTrace(System.err);              \_serviceUp = false;          } finally {              if(\_latch != null) {                  \_latch.countDown();              }          }      }        public String getServiceName() {          return \_serviceName;      }        public boolean isServiceUp() {          return \_serviceUp;      }      //This methos needs to be implemented by all specific service checker      public abstract void verifyService();  } |

**NetworkHealthChecker.java** : This class extends BaseHealthChecker and needs to provide only implementation of verifyService() method. **DatabaseHealthChecker.java** and **CacheHealthChecker.java** are same as NetworkHealthChecker.java apart from their service names and sleep time.

|  |
| --- |
| public class NetworkHealthChecker extends BaseHealthChecker  {      public NetworkHealthChecker (CountDownLatch latch)  {          super("Network Service", latch);      }        @Override      public void verifyService()      {          System.out.println("Checking " + this.getServiceName());          try          {              Thread.sleep(7000);          }          catch (InterruptedException e)          {              e.printStackTrace();          }          System.out.println(this.getServiceName() + " is UP");      }  } |

**ApplicationStartupUtil.java** : This clas is main startup class which initilizes the latch and wait of this latch till all services are checked.

|  |
| --- |
| public class ApplicationStartupUtil  {      //List of service checkers      private static List<BaseHealthChecker> \_services;        //This latch will be used to wait on      private static CountDownLatch \_latch;        private ApplicationStartupUtil()      {      }        private final static ApplicationStartupUtil INSTANCE = new ApplicationStartupUtil();        public static ApplicationStartupUtil getInstance()      {          return INSTANCE;      }        public static boolean checkExternalServices() throws Exception      {          //Initialize the latch with number of service checkers          \_latch = new CountDownLatch(3);            //All add checker in lists          \_services = new ArrayList<BaseHealthChecker>();          \_services.add(new NetworkHealthChecker(\_latch));          \_services.add(new CacheHealthChecker(\_latch));          \_services.add(new DatabaseHealthChecker(\_latch));            //Start service checkers using executor framework          Executor executor = Executors.newFixedThreadPool(\_services.size());            for(final BaseHealthChecker v : \_services)          {              executor.execute(v);          }            //Now wait till all services are checked          \_latch.await();            //Services are file and now proceed startup          for(final BaseHealthChecker v : \_services)          {              if( ! v.isServiceUp())              {                  return false;              }          }          return true;      }  } |

Now you can write any test class to check the functionality of latch.

|  |
| --- |
| public class Main {      public static void main(String[] args)      {          boolean result = false;          try {              result = ApplicationStartupUtil.checkExternalServices();          } catch (Exception e) {              e.printStackTrace();          }          System.out.println("External services validation completed !! Result was :: "+ result);      }  }    Output in console:    Checking Network Service  Checking Cache Service  Checking Database Service  Database Service is UP  Cache Service is UP  Network Service is UP  External services validation completed !! Result was :: true |

**Difference between a CyclicBarrier and a CountDownLatch**

* A CountDownLatch can be used only once in a program(until it’s count reaches 0).
* A CyclicBarrier can be used again and again once all the threads in a barriers is released.

# =============== CyclicBarrier ===================

A CyclicBarrier is a synchronizer that allows a set of threads to wait for each other to reach a common execution point, also called a barrier.

|  |
| --- |
| //JAVA program to demonstrate execution on Cyclic Barrier    import java.util.concurrent.TimeUnit;  import java.util.concurrent.TimeoutException;  import java.util.concurrent.BrokenBarrierException;  import java.util.concurrent.CyclicBarrier;    class Computation1 implements Runnable  {      public static int product = 0;      public void run()      {          product = 2 \* 3;          try          {              Tester.newBarrier.await();          }          catch (InterruptedException | BrokenBarrierException e)          {              e.printStackTrace();          }      }  }    class Computation2 implements Runnable  {      public static int sum = 0;      public void run()      {          // check if newBarrier is broken or not          System.out.println("Is the barrier broken? - " + Tester.newBarrier.isBroken());          sum = 10 + 20;          try          {              Tester.newBarrier.await(3000, TimeUnit.MILLISECONDS);                // number of parties waiting at the barrier              System.out.println("Number of parties waiting at the barrier "+              "at this point = " + Tester.newBarrier.getNumberWaiting());          }          catch (InterruptedException | BrokenBarrierException e)          {              e.printStackTrace();          }          catch (TimeoutException e)          {              e.printStackTrace();          }      }  }      public class Tester implements Runnable  {      public static CyclicBarrier newBarrier = new CyclicBarrier(3);        public static void main(String[] args)      {          // parent thread          Tester test = new Tester();            Thread t1 = new Thread(test);          t1.start();      }      public void run()      {          System.out.println("Number of parties required to trip the barrier = "+          newBarrier.getParties());          System.out.println("Sum of product and sum = " + (Computation1.product +          Computation2.sum));            // objects on which the child thread has to run          Computation1 comp1 = new Computation1();          Computation2 comp2 = new Computation2();            // creation of child thread          Thread t1 = new Thread(comp1);          Thread t2 = new Thread(comp2);            // moving child thread to runnable state          t1.start();          t2.start();            try          {              Tester.newBarrier.await();          }          catch (InterruptedException | BrokenBarrierException e)          {              e.printStackTrace();          }            // barrier breaks as the number of thread waiting for the barrier          // at this point = 3          System.out.println("Sum of product and sum = " + (Computation1.product +          Computation2.sum));            // Resetting the newBarrier          newBarrier.reset();          System.out.println("Barrier reset successful");      }  } |

**Output:**

<Number of parties required to trip the barrier = 3

Sum of product and sum = 0

Is the barrier broken? - false

Number of parties waiting at the barrier at this point = 0

Sum of product and sum = 36

Barrier reset successful

**Explanation:** The value of (sum + product) = 0 is printed on the console because the child thread has’t yet ran to set the values of sum and product variable. Following this, (sum + product) = 36 is printed on the console because the child threads ran setting the values of sum and product. Furthermore, the number of waiting thread on the barrier reached 3, due to which the barrier then allowed all thread to pass and finally 36 was printed. The value of “Number of parties waiting at the barrier at this point” = 0 because all the three threads had already called await() method and hence, the barrier is no longer active. In the end, newBarrier is reset and can be used again.