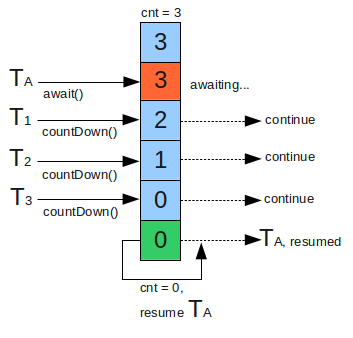
**Java concurrency – CountDownLatch Example**

As per java docs, **[CountDownLatch](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html" \o "CountDownLatch)** is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. CountDownLatch concept is very common [**interview question**](https://howtodoinjava.com/category/java/interview/http:/) in [**java concurrency**](https://howtodoinjava.com/category/java/multi-threading/), so make sure you understand it well. In this post, I will cover following points related to CountDownLatch in java concurrency.

**What is CountDownLatch?**

CountDownLatch was **introduced with JDK 1.5 along with other concurrent utilities like CyclicBarrier, Semaphore, [ConcurrentHashMap](https://howtodoinjava.com/java/collections/best-practices-for-using-concurrenthashmap/" \o "Best practices for using ConcurrentHashMap) and [BlockingQueue](https://howtodoinjava.com/java-5/how-to-use-blockingqueue-and-threadpoolexecutor-in-java/" \o "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.

CountDownLatch Concept

Pseudo code for CountDownLatch can be written like this:

**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 goind 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.

**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 |

**Common interview questions**

**Countdown latch:** A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads complete. Manager thread

Manager divided modules between dev teams A n B and he wants to assign it to QA team for testing only when both the teams completes their task.

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

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

Consider same scenario where manager divided modules between dev teams A n B. He goes on leave. He asked both the teams to wait for each other to complete their respective task and once both teams are done , assign it to QA team for testing.

Here manager thread works as main thread and development team works as worker thread. Development team threads wait for other development team threads after completing their task.

**Reusability**: Countdown latch instance cannot be used but cyclic barrier can be reused after all the waiting threads are released.