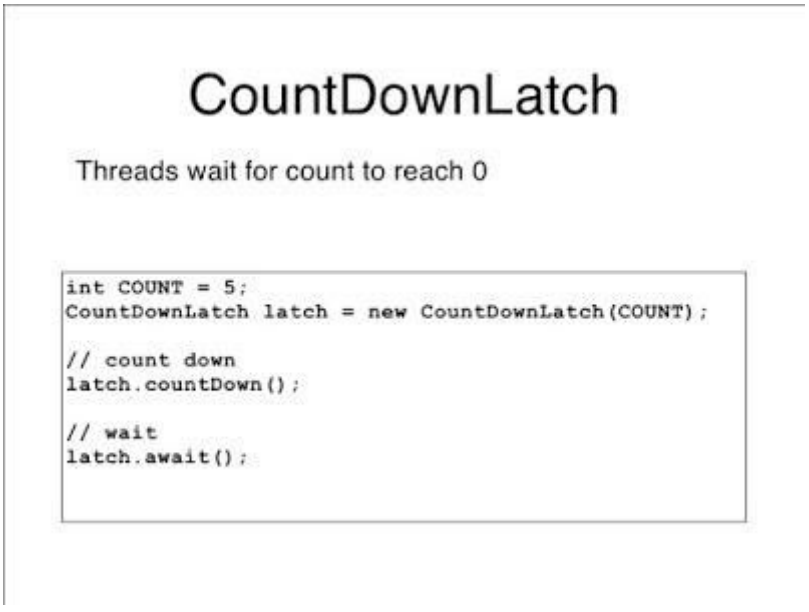
Countdown Latch

CountDowaLatch is a high-level synchronization utility which is used to prevent a particular thread to start processing until all threads are ready. This is achieved by a countdown. The thread, which needs to wait for starts with a counter, each thread them make the count down by 1 when they become ready, once the last thread call `countDown()` method, then the latch is broken and the thread waiting with counter starts running. `CountDownLatch` is a useful synchronizer and used heavily in multi-threaded testing. You can use this class to simulate truly concurrent behavior i.e. trying to access something at the same time once every thread is ready. Worth noting is that **`CountDownLatch` starts with a fixed number of counts** which cannot be changed later, though this restriction is re-mediated in Java 7 by introducing a similar but flexible concurrency utility called `Phaser`.

There is another similar utility called `CyclicBarrier`, which can also be used in this situation, where one thread needs to wait for other threads before they start processing. Only difference between `CyclicBarrier` and `CountDownLatch` is that you can reuse the barrier even after its broker but you cannot reuse the count down latch, once count reaches to zero.

How to use CountDownLatch in Java?

Theory is easy to read but you cannot understand it until you see it live in action, this is even more true with concurrency and multi-threading. So let's see how to use `CountDownLatch` in Java with a simple demo. In this example, we have a main thread which is required to wait until all worker thread finished their task. In order to achieve this, I have created a `CountDownLatch` with number of count equal to 4, which is the total number of worker threads. I then passed this `CountDownLatch` to each worker thread, whenever they complete their task, they call `countDown()` method, once last worker thread calls the `countDown()` method then the latch is broken and main thread which has been waiting on latch start running again and finished its execution.



In order to truly understand this problem, you first need to run by by commenting `latch.await()` call in main method, without this call main thread will not for any worker thread to finish their execution and it will terminate as soon as possible, may be even before any worker thread get started. If you run again by un-commenting `latch.await()` then you will always see that main thread has finished last. Why? because `await()` is a blocking call and it blocks until count reaches zero.

Also, you cannot reuse the latch once `count=0`, calling `await()` method on latch will have no effect i.e. thread will not block, but they not throw any exception as well, which is little bit of counter intuitive if you are expecting an IllegalThreadStateException

mportant points about CountDownLatch in Java

Let's revisit some important things about `CountDownLatch` in Java. This will help you to retain the knowledge you have just learned :

1) When you create an object of `CountDownLatch` you pass an int to its constructor (the count), this is actually number of invited parties (threads) for an event.

2) The thread, which is dependent on other threads to start processing, waits on latch until every other thread has called count down. All threads, which are waiting on `await()` proceed together once count down reaches to zero.

3) `countDown()` method decrements the count and `await()` method blocks until count == 0

4) Once count reaches to zero, *countdown latch cannot be used again*, calling `await()` method on that latch will not stop any thread, but it will neither throw any exception.

5) One of the popular use of `CountDownLatch` is in testing concurrent code, by using this latch you can guarantee that multiple threads are firing request simultaneously or executing code at almost same time.

6) There is a similar concurrency utility called `CyclicBarrier`, which can also use to simulate this scenario but difference between `CountDownLatch` and `CyclicBarrier` is that you can reuse the cyclic barrier once the barrier is broken but you cannot reuse the `CountDownLatch` one count reaches to zero.

7) Java 7 also introduced a flexible alternative of `CountDownLatch`, known as `Phaser`. It also has number of unarrived party just like barrier and latch but that number is flexible. So its more like you will not block if one of the guest bunks the party.