Items to work on

* What is this?
* Why do we specifically need this, not others?
* When do we use?
* How is implemented?

Java CyclicBarrier vs CountDownLatch

1. both ***CountDownLatch* and *CyclicBarrier* are used for managing multi-threaded applications**.
2. And, **they are both intended to express how a given thread or group of threads should wait.**

## Tasks vs.Threads

Let's take a deeper dive into some of the semantic differences between these two classes.

As stated in the definitions, CyclicBarrier allows a number of threads to wait on each other, whereas CountDownLatch allows one or more threads to wait for a number of tasks to complete.

In short, **CyclicBarrier maintains a count of threads** whereas **CountDownLatch maintains a count of tasks**.

### Cyclic Barrier(Thread)

### Example: Picnic Group Waiting a point

A CyclicBarrier is a reusable construct where a group of threads waits together until all of the threads arrive. At that point, the barrier is broken and an action can optionally be taken.

We can think of this like a group of friends. Every time they plan to eat at a restaurant they decide a common point where they can meet. They wait for each other there, and only when everyone arrives can they go to the restaurant to eat together.

CountDownLatch (TASK)

Latch Count down by anyone at anytime to meet the condition

Primary focus on Task

A CountDownLatch is a construct that a thread waits on while other threads count down on the latch until it reaches zero.

We can think of this like a dish at a restaurant that is being prepared. No matter which cook prepares however many of the n items, the waiter must wait until all the items are on the plate. If a plate takes n items, any cook will count down on the latch for each item she puts on the plate.

Note:

For simple use cases - services starting etc... a CountdownLatch is fine. A CyclicBarrier is useful for more complex co-ordination tasks. An example of such a thing would be parallel computation - where multiple subtasks are involved in the computation - kind of like [MapReduce](http://en.wikipedia.org/wiki/MapReduce).

Right - that's the major difference: CountDownLatch-->NumberOfCalls, CyclicBarrier-->NumberOfThreads –

**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.

In countDownLatch, there is one or more threads, that are waiting for a set of **other threads** to complete. In this situation, there are two types of threads, one type is waiting, another type is doing something, after finishes their tasks, they could be waiting or just terminated.

In CyclicBarrier, there are only one type of threads, they are waiting for each other, they are equal.

On a different note, CountDownLatch's inner class subclasses AQS (AbstractQueuedSynchronizer), while CyclicBarrier uses ReentrantLock

In **CountDownLatch**, main threads (However not necessarily, it can be any thread) waits for other threads to complete their execution. In **CyclicBarrier**, worker threads wait for each other to complete their execution.

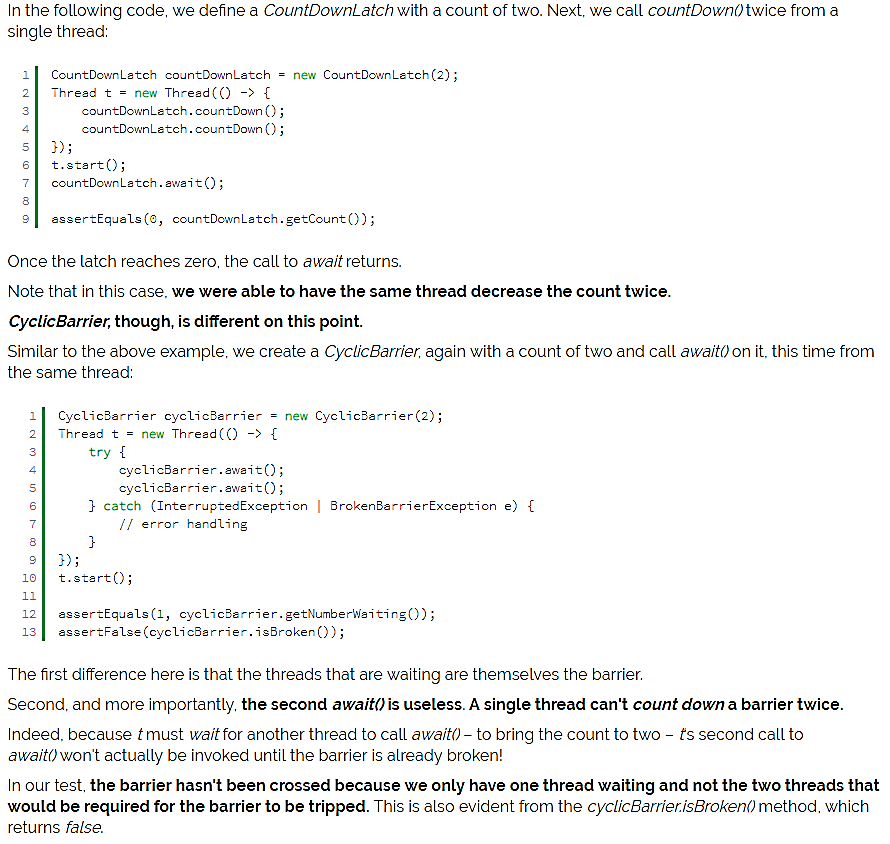
|  |  |
| --- | --- |
|  |  |
| When using a CyclicBarrier, the assumption is that you specify the number of waiting threads that trigger the barrier. If you specify 5, you must have at least 5 threads to call await(). | When using a CountDownLatch, you specify the number of calls to countDown() that will result in all waiting threads being released. This means that you can use a CountDownLatch with only a single thread. |
| As per Java Concurrency in Practice  barriers are for waiting for other threads. | Latches are for waiting for events; |
| One point that nobody has yet mentioned is that, in a CyclicBarrier, if a thread has a problem (timeout, interrupted...), all the others that have reached await() get an exception. via BrokenBarrierException (or InterruptedException |  |
| On a normal day, employee come to meeting wait for other to show up and if some attendees don`t come they have to wait indefinitely! in some special meeting the boss comes and he does not like to wait.(5 persons need to start meeting but only boss comes and also an enthusiastic employee) so he cancels the meeting (angrily) | **Latches**: if the angry boss wants to hold an exhibition for company customers, every thing needs to be ready (resources). we provide a to-do list every worker (Thread) dose his job and we check the to-do list (some workers do painting, others prepare sound system ...). when all the items in to-do list are complete (resources are provided) we can open the doors to customers. |
| CyclicBarrier.await() method which is a [blocking method in Java](http://javarevisited.blogspot.sg/2012/02/what-is-blocking-methods-in-java-and.html) and  blocks until all Thread or parties call await(). In general calling await() is shout out that Thread is waiting on the barrier. await() is a blocking call but can be timed out or Interrupted by other thread |  |
|  |  |

Cyclic Barrier (**Cyclic**)

1. As per name, its Cyclic so once reset it can be executed again.
2. One major difference is that [CyclicBarrier](http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/CyclicBarrier.html) takes an (optional) Runnable task which is run once the common barrier condition is met.
3. Once triggered the barrier is reset and can be used again.

**Task vs Thread**

When a Countdown Latch is constructed, we specify a count. For a typical use case, this count represents a number of tasks (or units of work) that need to be completed before the latch is released. The term ‘task’ is a generic term, not an explicit Java construct.



Must Read

<https://stackoverflow.com/questions/6595835/resettable-countdownlatch?noredirect=1&lq=1>

<https://www.baeldung.com/java-countdown-latch>

<https://www.baeldung.com/java-cyclic-barrier>

<https://www.geeksforgeeks.org/java-util-concurrent-cyclicbarrier-java/>

<https://stackoverflow.com/questions/4168772/java-concurrency-countdown-latch-vs-cyclic-barrier>

<https://stackoverflow.com/questions/6595835/resettable-countdownlatch?noredirect=1&lq=1>

Suspicious Resettable Count Down Latch

Excellent Learning : <https://stackoverflow.com/questions/6595835/resettable-countdownlatch?noredirect=1&lq=1>

I'm not sure if this is fatally flawed but I recently had the same problem and solved it by simply instantiating a new CountDownLatch object each time I wanted to reset. Something like this:

Waiter:

bla();

latch.await();

//now the latch has counted down to 0

blabla();

CountDowner

foo();

latch.countDown();

//now the latch has counted down to 0

latch = new CountDownLatch(1);

Waiter.receiveReferenceToNewLatch(latch);

bar();

Obviously this is a heavy abstraction but thus far it has worked for me and doesn't require you to tinker with any class definitions.

[Uzebeckatrente](https://stackoverflow.com/users/4466012/uzebeckatrente)

**336**55 silver badges2121 bronze badges

* This may suffer from memory safety issues in multithreaded environments (different processor cores may see different CountDownLatch instances at any given time). You should wrap your CountDownLatch reference in an AtomicReference.