Since: 1.5

Module java.base

Package java.util.concurrent

Class CountDownLatch

Java.util.concurrent.CountDownLatch

public class CountDownLatch extends Object

CountDownLatch is a very simple yet very common utility for blocking until a given number of signals, events, or conditions hold.

Sometimes you will want a thread to wait until one or more events have occurred. To handle such a situation, the concurrent API supplies CountDownLatch.

A CountDownLatch is initially created with a count of the number of events that must occur before the latch is released. Each time an event happens, the count is decremented. When the count reaches zero, the latch opens.

What is CountDownLatch in Java?

Answer: - CountDownLatch in Java is a kind of synchronizer which allows one Thread to wait for one or more Threads before starts processing.

This is a very crucial requirement and often needed in server-side core Java applications and having this functionality built-in as CountDownLatch greatly simplifies the development.

Note: - You can also implement the same functionality **using the wait and notify mechanism in Java** but it requires a lot of code and getting it to write in the first attempt is tricky.

With CountDownLatch it can be done in just a few lines. CountDownLatch also allows flexibility on the number of threads for which the main thread should wait, it can wait for one thread or n number of threads, there is not much change on code.

How CountDownLatch works in Java?

Answer: - CountDownLatch works in latch principle, the main thread will wait until Gate is open. One thread waits for n number of threads specified while creating CountDownLatch in Java.

Any thread, usually the main thread of application, which calls **CountDownLatch.await()** will wait until count reaches zero or it's interrupted by another Thread. All other threads are required to do a count down by calling **CountDownLatch.countDown()** once they are completed or ready to the job. as soon as count reaches zero, Thread awaiting starts running.

What are the disadvantages of CountDownLatch?

Answer: - The disadvantages of CountDownLatch is that it's **not reusable once count reaches zero** you cannot use CountDownLatch anymore.

When should we use CountDownLatch in Java?

Answer: - Use CountDownLatch when one of Thread like main thread, require to wait for one or more thread to complete before its start doing the processing. A classic example of using CountDownLatch in Java is any server-side core Java application that uses services architecture, where multiple services are provided by multiple threads and the application cannot start processing until all services have started successfully.

Things to remember

- You cannot reuse CountDownLatch once the count is reaches zero, this is the main difference between CountDownLatch and **CyclicBarrier**, which is frequently asked in core Java interviews and multi-threading interviews.

- Main Thread waits on Latch by calling CountDownLatch.await() method while other thread calls **CountDownLatch.countDown()** to inform that they have completed.

Methods of CountDownLatch: -

- **void await (): -** Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted.
- **boolean await (long timeout, TimeUnit unit): -** Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted, or the specified waiting time elapses.
- **void countDown(): -** Decrements the count of the latch, releasing all waiting threads if the count reaches zero.
- **long getCount(): -** Returns the current count.
- **String toString(): -** Returns a string identifying this latch, as well as its state.

What is difference between void await () and boolean await (long timeout, TimeUnit unit)?

Answer: -

void await ()	**boolean await (long timeout, TimeUnit unit)**
Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted.	Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted, or the specified waiting time elapses.
If the current count is zero then this method returns immediately.	If the current count is zero then this method returns immediately with the value true.
If the current count is greater than zero then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of two things happen: • The count reaches zero due to invocations of the countDown() method; or	If the current count is greater than zero then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of three things happen: • The count reaches zero due to invocations of the countDown() method; or

• Some other thread interrupts the current thread.	• Some other thread interrupts the current thread; or • The specified waiting time elapses. If the count reaches zero then the method returns with the value true.
If the current thread: • has its interrupted status set on entry to this method; or • is interrupted while waiting, then **InterruptedException** is thrown and the current thread's interrupted status is cleared.	If the current thread: • has its interrupted status set on entry to this method; or • is interrupted while waiting, then **InterruptedException** is thrown and the current thread's interrupted status is cleared. If the specified **waiting time elapses**, then the value **false** is returned. If the time is less than or equal to zero, the method will not wait at all.