

# CountDownLatch

## CountDownLatch

`CountDownLatch` is a synchronization primitive that comes with the `java.util.concurrent` package. It can be used to block a single or multiple threads while other threads complete their operations.

A `CountDownLatch` object is initialized with the number of tasks/threads it is required to wait for. Multiple threads can block and wait for the `CountDownLatch` object to reach zero by invoking `await()`. Every time a thread finishes its work, the thread invokes `countDown()` which decrements the counter by 1. Once the count reaches zero, threads waiting on the `await()` method are notified and resume execution.

The counter in the `CountDownLatch` cannot be reset making the `CountDownLatch` object un reusable. A `CountDownLatch` initialized with a count of 1 serves as an on/off switch where a particular thread is simply waiting for its only partner to complete. Whereas a `CountDownLatch` object initialized with a count of N indicates a thread waiting for N threads to complete their work. However, a single thread can also invoke `countDown()` N times to unblock a thread more than once.

If the `CountDownLatch` is initialized with zero, the thread would not wait for any other thread(s) to complete. The count passed is basically the number of times `countDown()` must be invoked before threads can pass through `await()`. If the `CountDownLatch` has reached zero and `countDown()` is again invoked, the latch will remain released hence making no difference.

A thread blocked on `await()` can also be interrupted by another thread as long as it is waiting and the counter has not reached zero.

Let's take an example where a master thread waits for worker threads to complete their execution.

Two workers, A & B, are being executed concurrently (two back to back threads initiated) while the master thread waits for them to finish. Every time a worker completes execution, the counter in the `CountDownLatch` is decremented by 1. Once all the workers have completed execution, the counter reaches 0 and notifies the threads blocked on the `await()` method. Subsequently, the latch opens and allows the master thread to run.

```
/**
 * The worker thread that has to complete its tasks first
 */
public class Worker extends Thread
{
    private CountDownLatch countDownLatch;

    public Worker(CountDownLatch countDownLatch, String name) {
        super(name);
        this.countDownLatch = countDownLatch;
    }

    @Override
    public void run()
    {
        System.out.println("Worker " + Thread.currentThread().getName
        ()+" started");
        try
        {
            Thread.sleep(3000);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
        System.out.println("Worker " + Thread.currentThread().getName
        ()+" finished");

        //Each thread calls countDown() method on task completion.
        countDownLatch.countDown();
    }
}
```

```

}

/**
 * The master thread that has to wait for the worker to complete it
 * s operations first
 */
public class Master extends Thread
{
    public Master(String name)
    {
        super(name);
    }

    @Override
    public void run()
    {
        System.out.println("Master executed "+Thread.currentThread().
getName());
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
    }
}

/**
 * The main thread that executes both the threads in a particular ord
 * er
 */
public class Main
{
    public static void main(String[] args) throws InterruptedException
    {
        //Created CountdownLatch for 2 threads
        CountdownLatch countDownLatch = new CountdownLatch(2);

        //Created and started two threads
        Worker A = new Worker(countDownLatch, "A");
        Worker B = new Worker(countDownLatch, "B");
    }
}

```

```

        A.start();

        B.start();

        //When two threads(A and B)complete their tasks, they are returned (counter reached 0).
        countDownLatch.await();

        //Now execution of master thread has started
        Master D = new Master("Master executed");
        D.start();
    }
}

```

main.java

Worker.java

Master.java



```

public class Master extends Thread
{
    public Master(String name)
    {
        super(name);
    }

    @Override
    public void run()
    {
        System.out.println("Master executed "+Thread.currentThread().getName());
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
    }
}

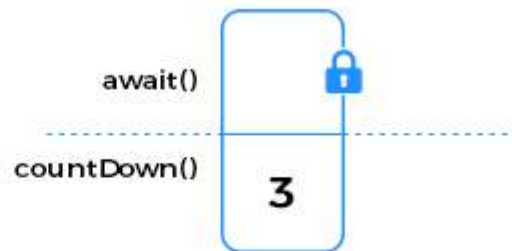
```



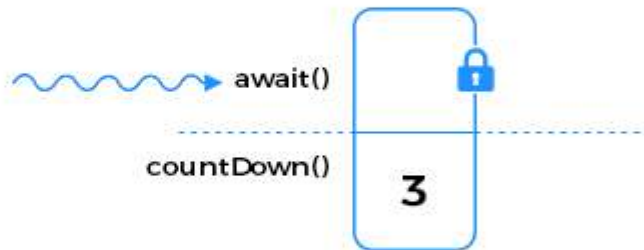
A pictorial representation appears below:

## CountDownLatch

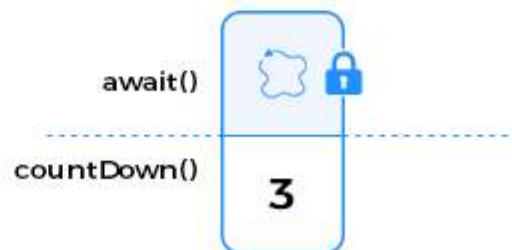
1. CountDownLatch initialized with a count of 3



2. A thread invokes await() on the CountDownLatch object



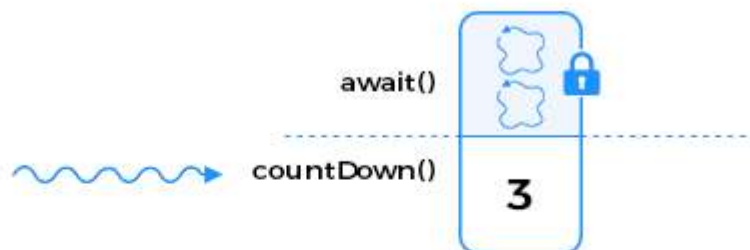
3. Thread is blocked till the count reaches 0



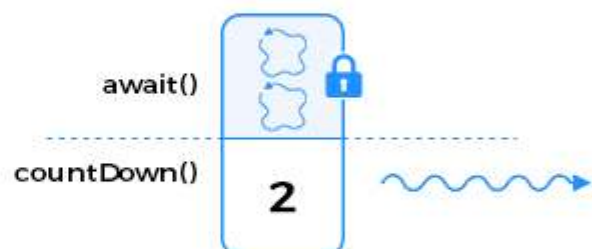
4. A second thread invokes await() and gets blocked



5. Two threads waiting for count to reach 0 and another thread invokes countDown()



6. Count is now 2



7. Another thread invokes



countDown()



countDown()

2

8. Count is decremented to 1

await()

countDown()

1



9. A third thread invokes countDown()



countDown()

1

10. Count reaches 0

await()

countDown()

0



11. Latch opens and the two blocked threads are allowed to proceed.

await()

countDown()

0

