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 unreusable. 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.

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().g
etName()+" started");
        try
        {
            Thread.sleep(3000);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
        System.out.println("Worker  "+Thread.currentThread().g
etName()+" finished");

        //Each thread calls countDown() method on task complet
ion.
        countDownLatch.countDown();
    }
}



/**
 * The master thread that has to wait for the worker to comple
```

```
te its operations first
 */
public class Master extends Thread
{
    public Master(String name)
    {
        super(name);
    }

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

    }
}

/**
 * The main thread that executes both the threads in a particu
lar order
 */
public class Main
{
    public static void main(String[] args) throws InterruptedE
xception
    {
        //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");
```

```
        Worker B = new Worker(countDownLatch, "B");


        A.start();
        B.start();

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

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

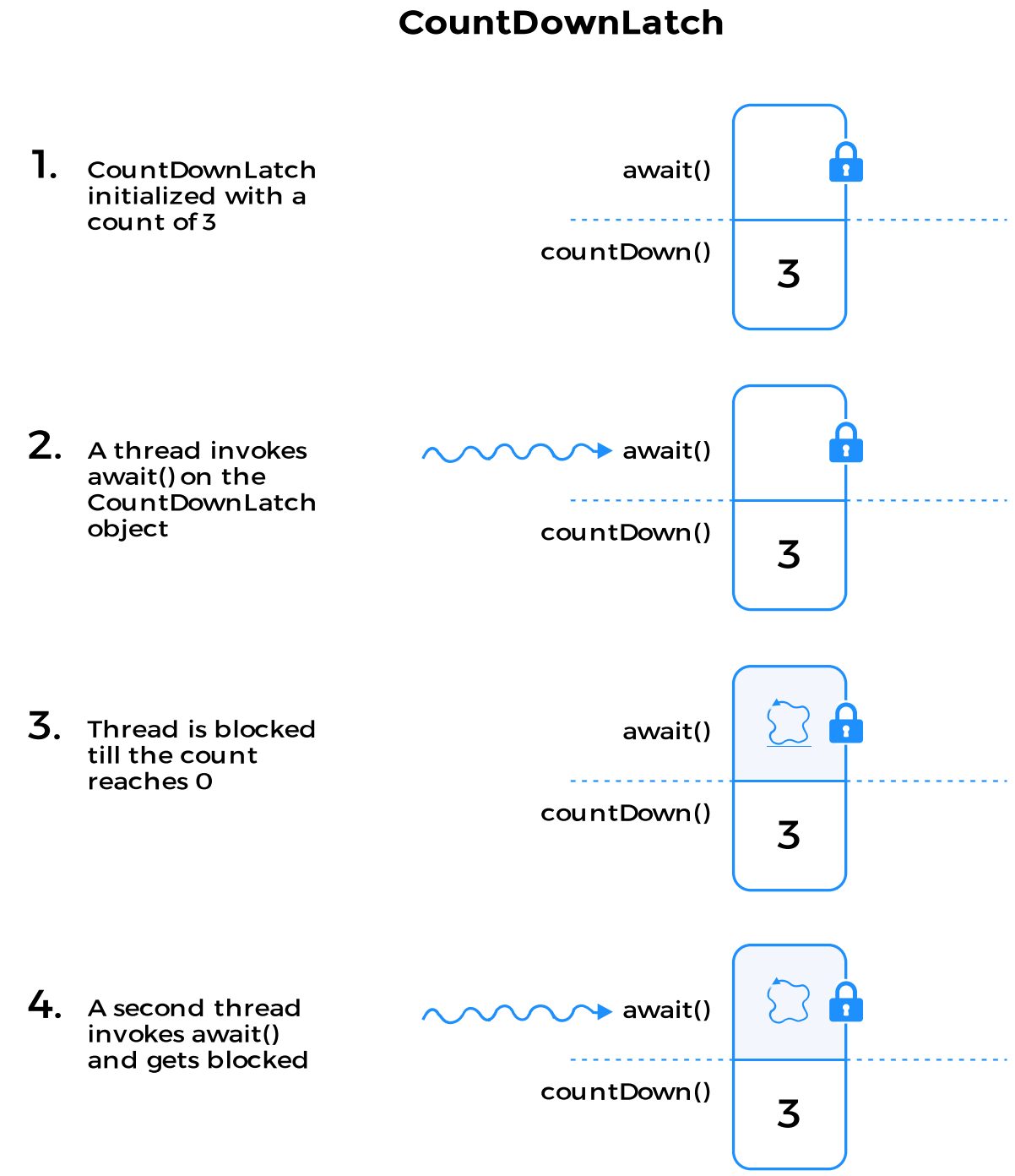
main.java

Worker.java

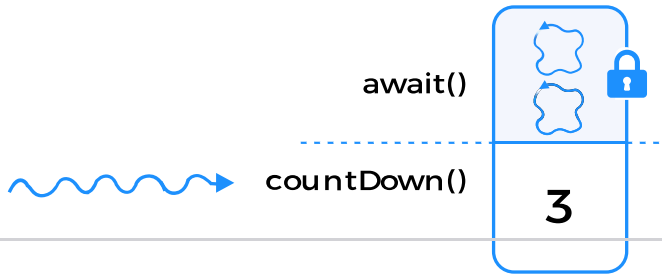
Master.java

```
1  import jav
2  public cla
3  {
4      publi
5      {
6
7          /
8          Co
9
10         /
11         Wo
12         Wo
13
14         A
15         B
16         co
17
18
19         /
20         co
21
22         /
23         Ma
24         D
25     }
26 }
```

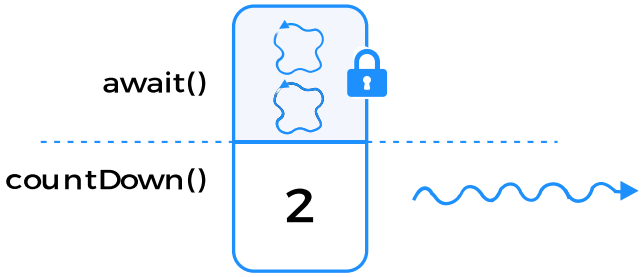
A pictorial representation appears below:



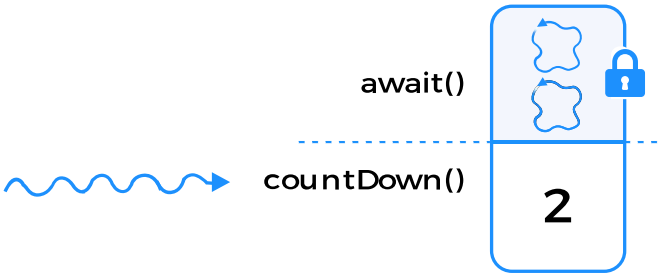
5. Two threads waiting for count to reach 0 and another thread invokes countDown()



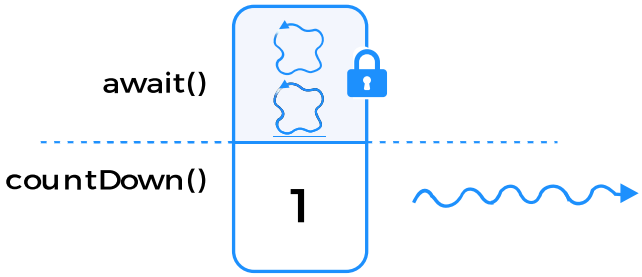
6. Count is now 2



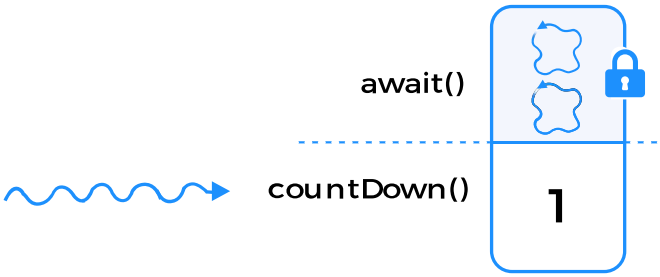
7. Another thread invokes countDown()



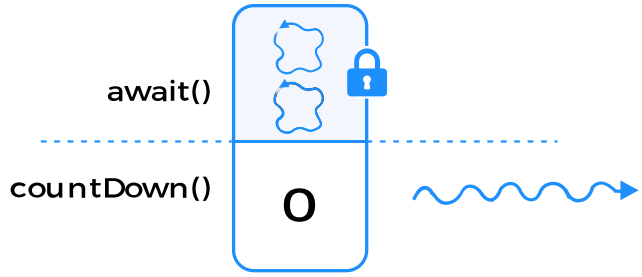
8. Count is decremented to 1



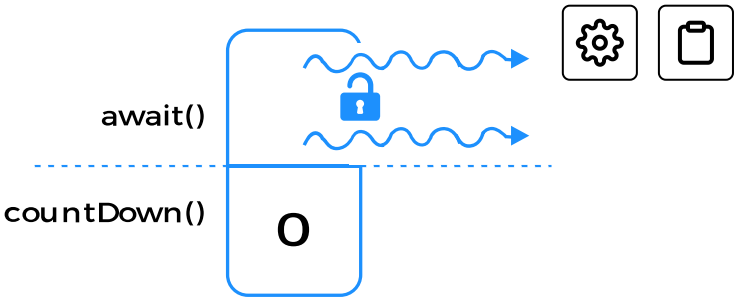
9. A third thread invokes countDown()



10. Count reaches 0



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



Back
ThreadLocal

Next
CyclicBarrier

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/countdownlatch__java-thread-basics__java-multithreading-for-senior-engineering-interviews)