| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/CountDownLatch.html) | [**Tree**](http://docs.google.com/package-tree.html) | [**Deprecated**](http://docs.google.com/deprecated-list.html) | [**Index**](http://docs.google.com/index-files/index-1.html) | [**Help**](http://docs.google.com/help-doc.html) | | --- | --- | --- | --- | --- | --- | --- | --- | | | ***Java™ Platform***  ***Standard Ed. 6*** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**PREV CLASS**](http://docs.google.com/java/util/concurrent/CopyOnWriteArraySet.html)   [**NEXT CLASS**](http://docs.google.com/java/util/concurrent/CyclicBarrier.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/CountDownLatch.html)    [**NO FRAMES**](http://docs.google.com/CountDownLatch.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |
| SUMMARY: NESTED | FIELD | [CONSTR](#3znysh7) | [METHOD](#2et92p0) | DETAIL: FIELD | [CONSTR](#3dy6vkm) | [METHOD](#4d34og8) |

## **java.util.concurrent**

Class CountDownLatch

[java.lang.Object](http://docs.google.com/java/lang/Object.html)  
 **java.util.concurrent.CountDownLatch**

public class **CountDownLatch**extends [Object](http://docs.google.com/java/lang/Object.html)

A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

A CountDownLatch is initialized with a given *count*. The [await](http://docs.google.com/java/util/concurrent/CountDownLatch.html#await()) methods block until the current count reaches zero due to invocations of the [countDown()](http://docs.google.com/java/util/concurrent/CountDownLatch.html#countDown()) method, after which all waiting threads are released and any subsequent invocations of [await](http://docs.google.com/java/util/concurrent/CountDownLatch.html#await()) return immediately. This is a one-shot phenomenon -- the count cannot be reset. If you need a version that resets the count, consider using a [CyclicBarrier](http://docs.google.com/java/util/concurrent/CyclicBarrier.html).

A CountDownLatch is a versatile synchronization tool and can be used for a number of purposes. A CountDownLatch initialized with a count of one serves as a simple on/off latch, or gate: all threads invoking [await](http://docs.google.com/java/util/concurrent/CountDownLatch.html#await()) wait at the gate until it is opened by a thread invoking [countDown()](http://docs.google.com/java/util/concurrent/CountDownLatch.html#countDown()). A CountDownLatch initialized to *N* can be used to make one thread wait until *N* threads have completed some action, or some action has been completed N times.

A useful property of a CountDownLatch is that it doesn't require that threads calling countDown wait for the count to reach zero before proceeding, it simply prevents any thread from proceeding past an [await](http://docs.google.com/java/util/concurrent/CountDownLatch.html#await()) until all threads could pass.

**Sample usage:** Here is a pair of classes in which a group of worker threads use two countdown latches:

* The first is a start signal that prevents any worker from proceeding until the driver is ready for them to proceed;
* The second is a completion signal that allows the driver to wait until all workers have completed.

class Driver { // ...  
 void main() throws InterruptedException {  
 CountDownLatch startSignal = new CountDownLatch(1);  
 CountDownLatch doneSignal = new CountDownLatch(N);  
  
 for (int i = 0; i < N; ++i) // create and start threads  
 new Thread(new Worker(startSignal, doneSignal)).start();  
  
 doSomethingElse(); // don't let run yet  
 startSignal.countDown(); // let all threads proceed  
 doSomethingElse();  
 doneSignal.await(); // wait for all to finish  
 }  
 }  
  
 class Worker implements Runnable {  
 private final CountDownLatch startSignal;  
 private final CountDownLatch doneSignal;  
 Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {  
 this.startSignal = startSignal;  
 this.doneSignal = doneSignal;  
 }  
 public void run() {  
 try {  
 startSignal.await();  
 doWork();  
 doneSignal.countDown();  
 } catch (InterruptedException ex) {} // return;  
 }  
  
 void doWork() { ... }  
 }

Another typical usage would be to divide a problem into N parts, describe each part with a Runnable that executes that portion and counts down on the latch, and queue all the Runnables to an Executor. When all sub-parts are complete, the coordinating thread will be able to pass through await. (When threads must repeatedly count down in this way, instead use a [CyclicBarrier](http://docs.google.com/java/util/concurrent/CyclicBarrier.html).)

class Driver2 { // ...  
 void main() throws InterruptedException {  
 CountDownLatch doneSignal = new CountDownLatch(N);  
 Executor e = ...  
  
 for (int i = 0; i < N; ++i) // create and start threads  
 e.execute(new WorkerRunnable(doneSignal, i));  
  
 doneSignal.await(); // wait for all to finish  
 }  
 }  
  
 class WorkerRunnable implements Runnable {  
 private final CountDownLatch doneSignal;  
 private final int i;  
 WorkerRunnable(CountDownLatch doneSignal, int i) {  
 this.doneSignal = doneSignal;  
 this.i = i;  
 }  
 public void run() {  
 try {  
 doWork(i);  
 doneSignal.countDown();  
 } catch (InterruptedException ex) {} // return;  
 }  
  
 void doWork() { ... }  
 }

Memory consistency effects: Actions in a thread prior to calling countDown() [*happen-before*](http://docs.google.com/package-summary.html#MemoryVisibility) actions following a successful return from a corresponding await() in another thread.

**Since:** 1.5

| **Constructor Summary** | |
| --- | --- |
| [**CountDownLatch**](http://docs.google.com/java/util/concurrent/CountDownLatch.html#CountDownLatch(int))(int count)            Constructs a CountDownLatch initialized with the given count. |

| **Method Summary** | |
| --- | --- |
| void | [**await**](http://docs.google.com/java/util/concurrent/CountDownLatch.html#await())()            Causes the current thread to wait until the latch has counted down to zero, unless the thread is [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()). |
| boolean | [**await**](http://docs.google.com/java/util/concurrent/CountDownLatch.html#await(long,%20java.util.concurrent.TimeUnit))(long timeout, [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit)            Causes the current thread to wait until the latch has counted down to zero, unless the thread is [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()), or the specified waiting time elapses. |
| void | [**countDown**](http://docs.google.com/java/util/concurrent/CountDownLatch.html#countDown())()            Decrements the count of the latch, releasing all waiting threads if the count reaches zero. |
| long | [**getCount**](http://docs.google.com/java/util/concurrent/CountDownLatch.html#getCount())()            Returns the current count. |
| [String](http://docs.google.com/java/lang/String.html) | [**toString**](http://docs.google.com/java/util/concurrent/CountDownLatch.html#toString())()            Returns a string identifying this latch, as well as its state. |

| **Methods inherited from class java.lang.**[**Object**](http://docs.google.com/java/lang/Object.html) |
| --- |
| [clone](http://docs.google.com/java/lang/Object.html#clone()), [equals](http://docs.google.com/java/lang/Object.html#equals(java.lang.Object)), [finalize](http://docs.google.com/java/lang/Object.html#finalize()), [getClass](http://docs.google.com/java/lang/Object.html#getClass()), [hashCode](http://docs.google.com/java/lang/Object.html#hashCode()), [notify](http://docs.google.com/java/lang/Object.html#notify()), [notifyAll](http://docs.google.com/java/lang/Object.html#notifyAll()), [wait](http://docs.google.com/java/lang/Object.html#wait()), [wait](http://docs.google.com/java/lang/Object.html#wait(long)), [wait](http://docs.google.com/java/lang/Object.html#wait(long,%20int)) |

| **Constructor Detail** |
| --- |

### CountDownLatch

public **CountDownLatch**(int count)

Constructs a CountDownLatch initialized with the given count.

**Parameters:**count - the number of times [countDown()](http://docs.google.com/java/util/concurrent/CountDownLatch.html#countDown()) must be invoked before threads can pass through [await()](http://docs.google.com/java/util/concurrent/CountDownLatch.html#await()) **Throws:** [IllegalArgumentException](http://docs.google.com/java/lang/IllegalArgumentException.html) - if count is negative

| **Method Detail** |
| --- |

### await

public void **await**()  
 throws [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html)

Causes the current thread to wait until the latch has counted down to zero, unless the thread is [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()).

If the current count is zero then this method returns immediately.

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()](http://docs.google.com/java/util/concurrent/CountDownLatch.html#countDown()) method; or
* Some other thread [interrupts](http://docs.google.com/java/lang/Thread.html#interrupt()) the current thread.

If the current thread:

* has its interrupted status set on entry to this method; or
* is [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()) while waiting,

then [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) is thrown and the current thread's interrupted status is cleared.

**Throws:** [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) - if the current thread is interrupted while waiting

### await

public boolean **await**(long timeout,  
 [TimeUnit](http://docs.google.com/java/util/concurrent/TimeUnit.html) unit)  
 throws [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html)

Causes the current thread to wait until the latch has counted down to zero, unless the thread is [interrupted](http://docs.google.com/java/lang/Thread.html#interrupt()), or the specified waiting time elapses.

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 three things happen:

* The count reaches zero due to invocations of the [countDown()](http://docs.google.com/java/util/concurrent/CountDownLatch.html#countDown()) method; or
* Some other thread [interrupts](http://docs.google.com/java/lang/Thread.html#interrupt()) 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](http://docs.google.com/java/lang/Thread.html#interrupt()) while waiting,

then [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) 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.

**Parameters:**timeout - the maximum time to waitunit - the time unit of the timeout argument **Returns:**true if the count reached zero and false if the waiting time elapsed before the count reached zero **Throws:** [InterruptedException](http://docs.google.com/java/lang/InterruptedException.html) - if the current thread is interrupted while waiting

### countDown

public void **countDown**()

Decrements the count of the latch, releasing all waiting threads if the count reaches zero.

If the current count is greater than zero then it is decremented. If the new count is zero then all waiting threads are re-enabled for thread scheduling purposes.

If the current count equals zero then nothing happens.

### getCount

public long **getCount**()

Returns the current count.

This method is typically used for debugging and testing purposes.

**Returns:**the current count

### toString

public [String](http://docs.google.com/java/lang/String.html) **toString**()

Returns a string identifying this latch, as well as its state. The state, in brackets, includes the String "Count =" followed by the current count.

**Overrides:**[toString](http://docs.google.com/java/lang/Object.html#toString()) in class [Object](http://docs.google.com/java/lang/Object.html) **Returns:**a string identifying this latch, as well as its state

| | [**Overview**](http://docs.google.com/overview-summary.html) | [**Package**](http://docs.google.com/package-summary.html) | **Class** | [**Use**](http://docs.google.com/class-use/CountDownLatch.html) | [**Tree**](http://docs.google.com/package-tree.html) | [**Deprecated**](http://docs.google.com/deprecated-list.html) | [**Index**](http://docs.google.com/index-files/index-1.html) | [**Help**](http://docs.google.com/help-doc.html) | | --- | --- | --- | --- | --- | --- | --- | --- | | | ***Java™ Platform***  ***Standard Ed. 6*** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**PREV CLASS**](http://docs.google.com/java/util/concurrent/CopyOnWriteArraySet.html)   [**NEXT CLASS**](http://docs.google.com/java/util/concurrent/CyclicBarrier.html) | [**FRAMES**](http://docs.google.com/index.html?java/util/concurrent/CountDownLatch.html)    [**NO FRAMES**](http://docs.google.com/CountDownLatch.html)     [**All Classes**](http://docs.google.com/allclasses-noframe.html) |
| SUMMARY: NESTED | FIELD | [CONSTR](#3znysh7) | [METHOD](#2et92p0) | DETAIL: FIELD | [CONSTR](#3dy6vkm) | [METHOD](#4d34og8) |

[Submit a bug or feature](http://bugs.sun.com/services/bugreport/index.jsp)

For further API reference and developer documentation, see [Java SE Developer Documentation](http://docs.google.com/webnotes/devdocs-vs-specs.html). That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.

Copyright 2006 Sun Microsystems, Inc. All rights reserved. Use is subject to [license terms](http://docs.google.com/legal/license.html). Also see the [documentation redistribution policy](http://java.sun.com/docs/redist.html).