Java Barrier Synchronizers: CountDownLatch (Part 1)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand how the Java CountDownLatch barrier synchronizer allows one or more threads to wait until a set of operations being performed in other threads completes.
Overview of Java
CountDownLatch
Overview of Java CountDownLatch

- Implements one (of several) Java barrier synchronizers

```java
public class CountDownLatch {
...
}
```

### Class CountDownLatch

```java
java.lang.Object
java.util.concurrent.CountDownLatch
```

```java
public class CountDownLatch extends Object
```

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` methods block until the current count reaches zero due to invocations of the `countDown()` method, after which all waiting threads are released and any subsequent invocations of `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`.

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` wait at the gate until it is opened by a thread invoking `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.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountDownLatch.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountDownLatch.html)
Overview of Java CountDownLatch

• Implements one (of several) Java barrier synchronizers

• Well-suited for fixed-size, one-shot “entry” & “exit” barriers

```
public class CountDownLatch {
...
}
```

Class CountDownLatch

java.lang.Object
  java.util.concurrent.CountDownLatch

```
public class CountDownLatch extends Object

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 methods block until the current count reaches zero due to invocations of the countDown() method, after which all waiting threads are released and any subsequent invocations of 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.

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 wait at the gate until it is opened by a thread invoking 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.
```
Overview of Java CountDownLatch

- Implements one (of several) Java barrier synchronizers
- Well-suited for fixed-size, one-shot “entry” & “exit” barriers

```java
public class CountDownLatch {
    ...
}
```

Class CountDownLatch

```java
java.lang.Object
does not implement an interface
```

```java
public class CountDownLatch extends Object
```

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 methods block until the current count reaches zero due to invocations of the countDown() method, after which all waiting threads are released and any subsequent invocations of 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.

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 wait at the gate until it is opened by a thread invoking 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.
Overview of Java CountDownLatch

- Applies a variant of *Bridge* pattern

```java
public class CountDownLatch {
    ...
}
```

See en.wikipedia.org/wiki/Bridge_pattern

Decouple the abstraction from the implementation so the two can vary independently.
Overview of Java CountDownLatch

- Applies a variant of *Bridge* pattern
- Locking handled by Sync
- Implementor hierarchy

```java
public class CountDownLatch {
    ...
    /** Performs sync mechanics */
    private final Sync sync;
    ...
}
Overview of Java CountDownLatch

- Applies a variant of *Bridge* pattern
- Locking handled by Sync Implementor hierarchy
- Inherits functionality from the AbstractQueuedSynchronizer class

```java
public class CountDownLatch {
    ...
    /** Performs sync mechanics */
    private final Sync sync;
    
    /**
     * Synchronization control or CountDownLatch. Uses AQS state to represent count.
     */
    private static final class Sync extends AbstractQueuedSynchronizer {
        ...
    }
}
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html
Key Methods in Java

CountDownLatch
• The constructor initializes CountDownLatch with count

```java
public class CountDownLatch {
    ...
    public CountDownLatch (int count) {
        ...
        this.sync = new Sync(count);
    }
    ...
}
```
Key Methods in Java CountDownLatch

• The constructor initializes CountDownLatch with count
• The count cannot be reset

```java
public class CountDownLatch {
    ... 
    public CountDownLatch (int count) {
        ... 
        this.sync = new Sync(count);
    }
    ...
}
```
public class CountDownLatch {
    ...
    public void await() {
        sync.acquireSharedInterruptibly(1);
    }

    public boolean await(long timeout, TimeUnit unit) {
        return sync.
            tryAcquireSharedNanos(1, unit.toNanos(timeout));
    }

    public void countDown() {
        sync.releaseShared(1);
    }
    ...
}
Key Methods in Java CountDownLatch

- Key methods countdown & await the countdown to reach 0

```java
public class CountDownLatch {
    ...
    public void await() ...
    {
        sync.acquireSharedInterruptibly(1);
    }

    public boolean await(
        long timeout,
        TimeUnit unit)
    ...
    {
        return sync.
            tryAcquireSharedNanos
                (1, unit.toNanos(timeout));
    }

    public void countDown()
    {
        sync.releaseShared(1);
    }
    ...
}
```

Methods forward to underlying AbstractQueuedSynchronizer methods
Key Methods in Java CountDownLatch

• Key methods `countdown` & `await`
  the countdown to reach 0
• Causes the calling thread to wait
  until the latch’s count reaches 0

```java
public class CountDownLatch {
    ...
    public void await() ... {
        sync.acquire...(1);
    }
    ...
}
```
Key Methods in Java CountDownLatch

- Key methods countdown & await the countdown to reach 0
  - Causes the calling thread to wait until the latch’s count reaches 0
  - Causes the calling thread to wait until latch counts down to 0
  - Unless thread is interrupted or waiting time elapses first

```java
public class CountDownLatch {
    ...
    public boolean await(
        long timeout, TimeUnit unit) ...
    {
        return sync.
            tryAcquireSharedNanos
                (1, unit.toNanos(timeout));
    }
    ...
```
Key Methods in Java CountDownLatch

- Key methods countdown & await the countdown to reach 0
  - Causes the calling thread to wait until the latch’s count reaches 0
  - Causes the calling thread to wait until latch counts down to 0
  - Decrements latch count – if count reaches 0 release waiting threads

```java
public class CountDownLatch {
    ...
    public void countDown() {
        sync.releaseShared(1);
    }
    ...
}
```

Threads that countDown() needn’t wait for count to reach 0 before proceeding
End of Java Barrier Synchronizers: CountDownLatch (Part 1)
Java Barrier Synchronizers: 
CountDownLatch (Part 2)

Douglas C. Schmidt  
d.schmidt@vanderbilt.edu  
www.dre.vanderbilt.edu/~schmidt

Institute for Software 
Integrated Systems  
Vanderbilt University  
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

• Understand how the Java CountDownLatch barrier synchronizer allows one or more threads to wait until a set of operations being performed in other threads completes.

• Know how to program with Java CountDownLatch in practice.

class GCDCountDownLatchTester implements Runnable {
    private final CountDownLatch mEntryBarrier;
    private final CountDownLatch mExitBarrier;
    ...

    GCDCountDownLatchTester(CountDownLatch entryBarrier,
                            CountDownLatch exitBarrier, ...) {
        mEntryBarrier = entryBarrier; mExitBarrier = exitBarrier;
        ...
    }

    public void run() {
        ...
        mEntryBarrier.await();
        runTest();
        mExitBarrier.countDown();
        ...
    }
}
Using the Java CountDownLatch in Practice
Using the Java CountDownLatch in Practice

- An app that uses CountDownLatch to coordinate the concurrent benchmarking of different Greatest Common Divisor (GCD) implementations, which compute the largest positive integer that is a divisor of two numbers

See github.com/douglasraigschmidt/POSA/tree/master/ex/M3/GCD/CountDownLatch
Using the Java CountDownLatch in Practice

- Create worker threads that use two CountDownLatchs as entry & exit barriers

```java
class GCDCountDownLatchTest {
    void main() throws InterruptedException {
        ... 
        List<GCDTuple> gcdTests = makeTests();

        CountDownLatch entryBarrier = new CountDownLatch(1);
        CountDownLatch exitBarrier =
            new CountDownLatch(gcdTests.size());

        for (GCDTuple gcdTest : gcdTests)
            new Thread(new GCDCountDownLatchTester
                (entryBarrier, exitBarrier, gcdTuple, this)).start();

        System.out.println("Starting tests");
        entryBarrier.countDown();
        System.out.println("Waiting for results");
        exitBarrier.await();
        System.out.println("All tests done"); ...
```
Using the Java CountDownLatch in Practice

• Create worker threads that use two CountDownLatchs as entry & exit barriers

```java
class GCDCountDownLatchTest {
    void main() throws InterruptedException {
        ...
        List<GCDTuple> gcdTests = makeTests();

        CountDownLatch entryBarrier = new CountDownLatch(1);
        CountDownLatch exitBarrier =
            new CountDownLatch(gcdTests.size());

        for (GCDTuple gcdTest : gcdTests)
            new Thread(new GCDCountDownLatchTester
                (entryBarrier, exitBarrier, gcdTestTuple, this)).start();

        System.out.println("Starting tests");
        entryBarrier.countDown();
        System.out.println("Waiting for results");
        exitBarrier.await();
        System.out.println("All tests done"); ...
    }
}
```

Create barrier synchronizers
Using the Java CountDownLatch in Practice

• Create worker threads that use two CountDownLatchs as entry & exit barriers

```java
class GCDCountDownLatchTest {
    void main() throws InterruptedException {
        ...
        List<GCDTuple> gcdTests = makeTests();

        CountDownLatch entryBarrier = new CountDownLatch(1);
        CountDownLatch exitBarrier =
            new CountDownLatch(gcdTests.size());

        for (GCDTuple gcdTest : gcdTests)
            new Thread(new GCDCountDownLatchTester
                (entryBarrier, exitBarrier, gcdTuple, this)).start();

        System.out.println("Starting tests");
        entryBarrier.countDown();
        System.out.println("Waiting for results");
        exitBarrier.await();
        System.out.println("All tests done"); ...
```
Using the Java CountDownLatch in Practice

- Create worker threads that use two CountDownLatchs as entry & exit barriers

```java
class GCDCountDownLatchTest {
    void main() throws InterruptedException {
        ...
        List<GCDTuple> gcdTests = makeTests();

        CountDownLatch entryBarrier = new CountDownLatch(1);
        CountDownLatch exitBarrier =
            new CountDownLatch(gcdTests.size());

        for (GCDTuple gcdTest : gcdTests)
            new Thread(new GCDCountDownLatchTester
                (entryBarrier, exitBarrier, gcdTuple, this)).start();

        System.out.println("Starting tests");  // Don't start yet
        entryBarrier.countDown();
        System.out.println("Waiting for results");
        exitBarrier.await();
        System.out.println("All tests done"); ...
    }
}
```
Using the Java CountDownLatch in Practice

• Create worker threads that use two CountDownLatchs as entry & exit barriers

class GCDCountDownLatchTest {
    void main() throws InterruptedException {
        ...  
        List<GCDTuple> gcdTests = makeTests();

        CountDownLatch entryBarrier = new CountDownLatch(1);
        CountDownLatch exitBarrier =
            new CountDownLatch(gcdTests.size());

        for (GCDTuple gcdTest : gcdTests)
            new Thread(new GCDCountDownLatchTester
                (entryBarrier, exitBarrier, gcdTuple, this)).start();

        System.out.println("Starting tests");
        entryBarrier.countDown();  Let all worker threads proceed
        System.out.println("Waiting for results");
        exitBarrier.await();
        System.out.println("All tests done"); ...

The countDown() method is a “latch” that let’s all the tester threads start running, but it doesn’t ensure all the tester threads start at the same time.
Using the Java CountDownLatch in Practice

• Create worker threads that use two CountDownLatchs as entry & exit barriers

```java
class GCDCountDownLatchTest {
    void main() throws InterruptedException {
        ...
        List<GCDTuple> gcdTests = makeTests();

        CountDownLatch entryBarrier = new CountDownLatch(1);
        CountDownLatch exitBarrier =
            new CountDownLatch(gcdTests.size());

        for (GCDTuple gcdTest : gcdTests)
            new Thread(new GCDCountDownLatchTester
                (entryBarrier, exitBarrier, gcdTuple, this)).start();

        System.out.println("Starting tests");
        entryBarrier.countDown();
        System.out.println("Waiting for results");
        exitBarrier.await(); // Wait for all to finish (exit barrier)
        System.out.println("All tests done"); ...
    }
}
```

After `await()` returns for a CountDownLatch it can’t be reused/reset without creating a new CountDownLatch instance
Using the Java CountDownLatch in Practice

- Create worker threads that use two CountDownLatchs as entry & exit barriers

Define a worker that runs in a thread

class GCDCountDownLatchTester implements Runnable {
    private final CountDownLatch mEntryBarrier;
    private final CountDownLatch mExitBarrier;
    ...

    GCDCountDownLatchTester(CountDownLatch entryBarrier,
                            CountDownLatch exitBarrier, ...) {
        mEntryBarrier = entryBarrier; mExitBarrier = exitBarrier;
        ...
    }

    public void run() {
        ...
        mEntryBarrier.await();
        runTest();
        mExitBarrier.countDown();
        ...
    }
}
Using the Java CountDownLatch in Practice

• Create worker threads that use two CountDownLatchs as entry & exit barriers

```java
class GCDCountDownLatchTester implements Runnable {
    private final CountDownLatch mEntryBarrier;
    private final CountDownLatch mExitBarrier;
    ...

    GCDCountDownLatchTester(CountDownLatch entryBarrier,
                            CountDownLatch exitBarrier, ...) {
        mEntryBarrier = entryBarrier; mExitBarrier = exitBarrier;
        ...
    }

    public void run() {
        ...
        mEntryBarrier.await();
        runTest();
        mExitBarrier.countDown();
        ...
    }
```

Initialize barrier fields
Using the Java CountDownLatch in Practice

- Create worker threads that use two CountDownLatchs as entry & exit barriers

```java
class GCDCountDownLatchTester implements Runnable {
    private final CountDownLatch mEntryBarrier;
    private final CountDownLatch mExitBarrier;
    ...

    GCDCountDownLatchTester(CountDownLatch entryBarrier,
                            CountDownLatch exitBarrier, ...) {
        mEntryBarrier = entryBarrier; mExitBarrier = exitBarrier;
        ...
    }

    public void run() {
        ...
        mEntryBarrier.await();  // Wait until main thread is ready (entry barrier)
        runTest();
        mExitBarrier.countDown();
        ...
    }
}
```
Using the Java CountDownLatch in Practice

- Create worker threads that use two CountDownLatchs as entry & exit barriers

```java
class GCDCountDownLatchTester implements Runnable {
    private final CountDownLatch mEntryBarrier;
    private final CountDownLatch mExitBarrier;
    ...

    GCDCountDownLatchTester(CountDownLatch entryBarrier,
                            CountDownLatch exitBarrier, ...) {
        mEntryBarrier = entryBarrier; mExitBarrier = exitBarrier;
    }

    public void run() {
        ...
        mEntryBarrier.await();
        runTest();  // Do some processing
        mExitBarrier.countDown();
        ...
    }
}
```
Using the Java CountDownLatch in Practice

- Create worker threads that use two CountDownLatchs as entry & exit barriers

```java
class GCDCountDownLatchTester implements Runnable {
    private final CountDownLatch mEntryBarrier;
    private final CountDownLatch mExitBarrier;
    ...

    GCDCountDownLatchTester(CountDownLatch entryBarrier,
                           CountDownLatch exitBarrier, ...) {
        mEntryBarrier = entryBarrier; mExitBarrier = exitBarrier;
        ...
    }

    public void run() {
        ...
        mEntryBarrier.await();
        runTest();
        runTest();
        mExitBarrier.countDown();  // Let main thread proceed
        ...
    }
}
```
End of Java Barrier Synchronizers: CountDownLatch (Part 2)
1. Which of the following are consequences of the `countDown()` method letting all the tester threads start running, but not ensuring all the tester threads start at the same time..

   a. Threads that start earlier will get a head start on threads that start later.
   b. The `GCDCountDownLatchTest` program won’t exit until all the tester threads are finished running
   c. After `await()` returns for a `CountDownLatch` it can’t be reused/reset without creating a new `CountDownLatch` instance
   d. A `CountDownLatch` can only be used with a fixed number of threads