How to Leverage Cyclic Barriers

A CyclicBarrier is a Java synchronization class that - along with Phaser and CountdownLatch - enqueue threads.

All three classes can be found in the same package:

```
java.util.concurrent
```

While this topic encompasses a simple concept, its implementation can be involved (when observing the Separation of Concerns design pattern, so I will eventually supplement the information conveyed in this article with a Sequence Diagram or anything else that provides clarity.

A use case for a synchronization barrier is to bench test (using JMH, of course) different tool.

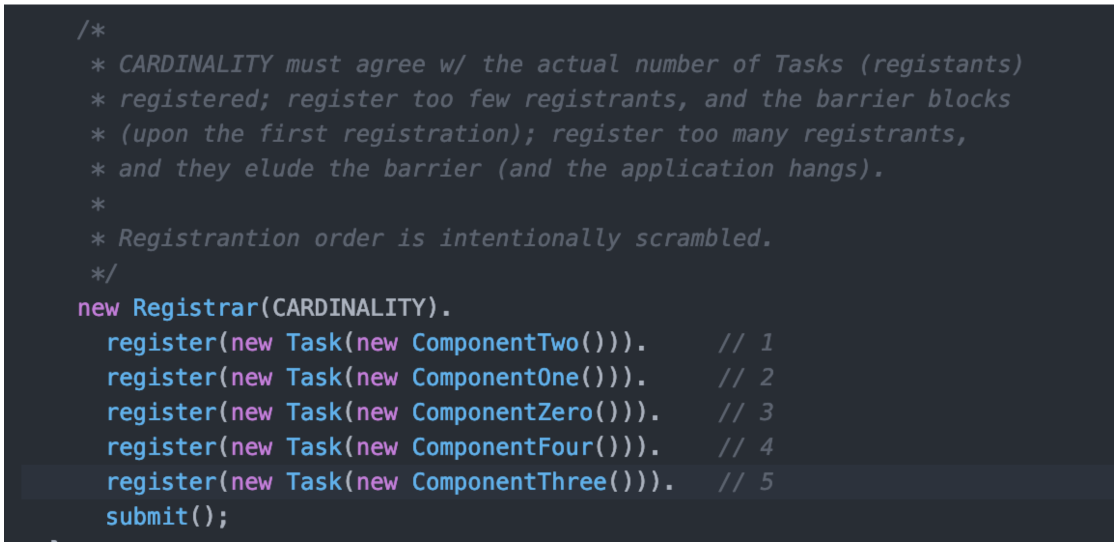
One specific use case is testing the responsiveness of various time-series databases.

Below, an illustration of the output of the Cyclic Barrier demo application is shown.

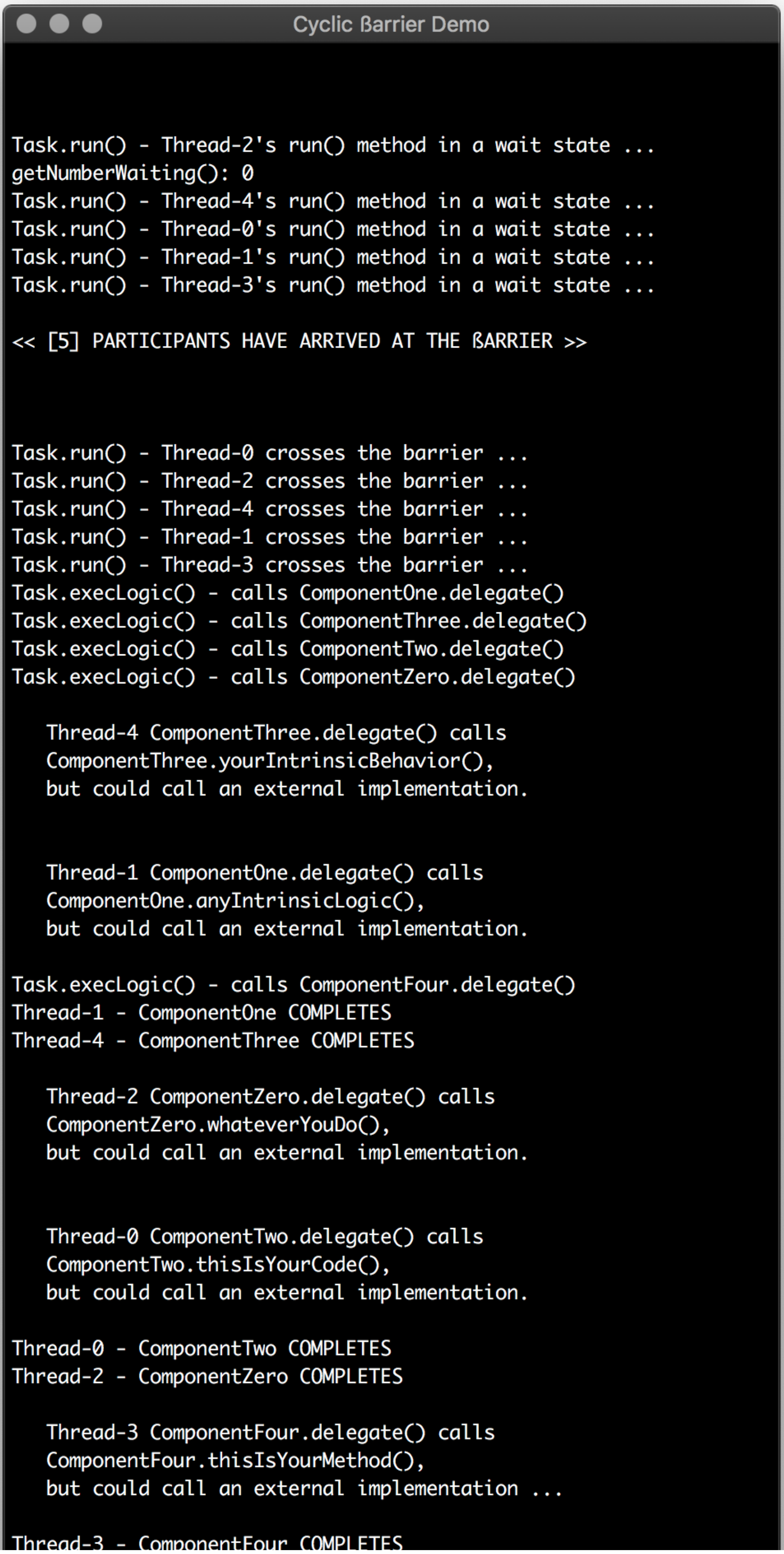
The async nature of threads means that the order of events differs each time the application is invoked.

Participating threads are initially in a wait state, awaiting all other threads' registration with the CyclicBarrier instance.

As can be seen below, the true order of registration is under the influence of the JVM's ThreadScheduler:



```
    /*
     * CARDINALITY must agree w/ the actual number of Tasks (registants)
     * registered; register too few registrants, and the barrier blocks
     * (upon the first registration); register too many registrants,
     * and they elude the barrier (and the application hangs).
     *
     * Registrantion order is intentionally scrambled.
     */
    new Registrar(CARDINALITY).
      register(new Task(new ComponentTwo())).     // 1
      register(new Task(new ComponentOne())).     // 2
      register(new Task(new ComponentZero())).    // 3
      register(new Task(new ComponentFour())).    // 4
      register(new Task(new ComponentThree())).   // 5
      submit();
```


Cyclic Barrier Demo

```
Task.run() - Thread-2's run() method in a wait state ...
getNumberWaiting(): 0
Task.run() - Thread-4's run() method in a wait state ...
Task.run() - Thread-0's run() method in a wait state ...
Task.run() - Thread-1's run() method in a wait state ...
Task.run() - Thread-3's run() method in a wait state ...

<< [5] PARTICIPANTS HAVE ARRIVED AT THE BARRIER >>



Task.run() - Thread-0 crosses the barrier ...
Task.run() - Thread-2 crosses the barrier ...
Task.run() - Thread-4 crosses the barrier ...
Task.run() - Thread-1 crosses the barrier ...
Task.run() - Thread-3 crosses the barrier ...
Task.execLogic() - calls ComponentOne.delegate()
Task.execLogic() - calls ComponentThree.delegate()
Task.execLogic() - calls ComponentTwo.delegate()
Task.execLogic() - calls ComponentZero.delegate()

   Thread-4 ComponentThree.delegate() calls
   ComponentThree.yourIntrinsicBehavior(),
   but could call an external implementation.


   Thread-1 ComponentOne.delegate() calls
   ComponentOne.anyIntrinsicLogic(),
   but could call an external implementation.

Task.execLogic() - calls ComponentFour.delegate()
Thread-1 - ComponentOne COMPLETES
Thread-4 - ComponentThree COMPLETES

   Thread-2 ComponentZero.delegate() calls
   ComponentZero.whateverYouDo(),
   but could call an external implementation.


   Thread-0 ComponentTwo.delegate() calls
   ComponentTwo.thisIsYourCode(),
   but could call an external implementation.

Thread-0 - ComponentTwo COMPLETES
Thread-2 - ComponentZero COMPLETES

   Thread-3 ComponentFour.delegate() calls
   ComponentFour.thisIsYourMethod(),
   but could call an external implementation ...

Thread-3 - ComponentFour COMPLETES
```

```
~/stage/hold/cyclicbarrier
```

As you can observe above, the order in which each thread is given time (to delegate), is still subject to the JVM's ThreadScheduler.

Additionally, with a larger number of competing threads, each of which perform a wider variance of more elaborate/intensive processing, the intervals and swapping would be more pronounced.

Finally, the order of completion is (in this example), is more dependent upon the JVM's ThreadScheduler (than any complexity in the individual workers /threads).

---

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html

https://en.wikipedia.org/wiki/Barrier_(computer_science)