Java Barrier Synchronizers: CyclicBarrier (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 CyclicBarrier synchronizer allows a set of threads to all wait for each other to reach a common barrier point.

<<Java Class>>

CyclicBarrier

- CyclicBarrier(int, Runnable)
- CyclicBarrier(int)
- getParties(): int
- await(): int
- await(long, TimeUnit): int
- isBroken(): boolean
- reset(): void
- getNumberOfWaiting(): int
Overview of Java

CyclicBarrier
Overview of Java CyclicBarrier

- Implements another Java barrier synchronizer

```java
public class CyclicBarrier {
    ...
}
```

Class CyclicBarrier

```
java.lang.Object
    java.util.concurrent.CyclicBarrier

public class CyclicBarrier
    extends Object
```

A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released.

A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This barrier action is useful for updating shared-state before any of the parties continue.

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

- Implements another Java barrier synchronizer
- Well-suited for fixed-size “cyclic” & “entry” and/or “exit” barriers

Class CyclicBarrier

```java
public class CyclicBarrier {
    ...
}
```

A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released.

A `CyclicBarrier` supports an optional `Runnable` command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This *barrier action* is useful for updating shared-state before any of the parties continue.

Useful for fixed-sized number of threads that wait for all threads to complete before proceeding
Overview of Java CyclicBarrier

- Implements another Java barrier synchronizer
  - Well-suited for fixed-size “cyclic” & “entry” and/or “exit” barriers
  - Enables count to be reset at any point

**Class CyclicBarrier**

```java
public class CyclicBarrier {
    ...
}
```

A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released.

A **CyclicBarrier** supports an optional **Runnable** command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This **barrier action** is useful for updating shared-state before any of the parties continue.
Overview of Java CyclicBarrier

- Does not apply the *Bridge* pattern

```java
public class CyclicBarrier {
    ...
}
```
Overview of Java CyclicBarrier

- Does not apply the *Bridge* pattern
- Nor does it implement any Java interfaces or use the Abstract QueuedSynchronizer framework

```java
public class CyclicBarrier {
    ...
}
```
Key Methods in Java
CyclicBarrier
Overview of Java CyclicBarrier

- Constructor initializes object to trip when given # of parties wait on it

```java
public class CyclicBarrier {
    ...
    public CyclicBarrier (int parties) {
    }

    public CyclicBarrier (int parties,
        Runnable barrierAction) {
        ...
    }
    ...
}
```
Overview of Java CyclicBarrier

- Constructor initializes object to trip when given # of parties wait on it

```java
public class CyclicBarrier {
    ...
    public CyclicBarrier (int parties) {
    }
    ...
    public CyclicBarrier (int parties,
        Runnable barrierAction) {
        ...
    }
    ...
}
```

“Parties” == “Threads”

Requires a fixed number of threads that is identical to the number of parties.
Overview of Java CyclicBarrier

- Constructor initializes object to trip when given # of parties wait on it
- Optionally given a barrier action to execute when barrier’s tripped

```java
public class CyclicBarrier {
    ...
    public CyclicBarrier (int parties) {
        ...
    }

    public CyclicBarrier (int parties,
                        Runnable barrierAction) {
        ...
    }

    ...
}```
Overview of Java CyclicBarrier

- Constructor initializes object to trip when given # of parties wait on it
  - Optionally given a *barrier action* to execute when barrier’s tripped
  - Performed by the last thread entering the barrier

```java
public class CyclicBarrier {
    ...
    public CyclicBarrier (int parties) {
    }
    ...
    public CyclicBarrier (int parties,
            Runnable barrierAction) {
        ...
    }
}

Parties are suspended when it is executed
Overview of Java CyclicBarrier

- Constructor initializes object to trip when given # of parties wait on it
- Optionally given a *barrier action* to execute when barrier’s tripped
  - Performed by the last thread entering the barrier
- Useful for updating shared-state before any parties continue

```java
class CyclicBarrier {
    ...
    public CyclicBarrier (int parties) {
        ...
    }
    public CyclicBarrier (int parties,
                           Runnable barrierAction) {
        ...
    }
}
```
Overview of Java CyclicBarrier

• Constructor initializes object to trip when given # of parties wait on it
• Optionally given a barrier action to execute when barrier’s tripped
  • Performed by the last thread entering the barrier
• Useful for updating shared-state before any parties continue
• Count is reset to initial number of parties after barrier’s tripped

```java
public class CyclicBarrier {
  ...
  public CyclicBarrier {
    ...
    public CyclicBarrier (int parties) {
      ...
      public CyclicBarrier
        (int parties,
         Runnable barrierAction) {
        ...
      }
    }
```

...
Overview of Java CyclicBarrier

- Methods wait until all parties call `await()` on barrier & then reset it

```java
public class CyclicBarrier {
    ...
    public int await() { ... }

    public int await(long timeout,
                     TimeUnit unit)
    { ... }

    public void reset() { ... }
    ...
}
```
Overview of Java CyclicBarrier

- Methods wait until all parties call `await()` on barrier & then reset it
- Block until all parties arrive, barrier resets, or thread interrupted

```java
class CyclicBarrier {
    ...
    public int await() {
        ...
    }
    ...
}
```
Overview of Java CyclicBarrier

- Methods wait until all parties call `await()` on barrier & then reset it
- Block until all parties arrive, barrier resets, or thread interrupted
- Returns arrival index of the thread at the barrier

```java
public class CyclicBarrier {
    ...
    public int await() { ... }
    ...
}
```

```
if (barrier.await() == 0) {
    // log completion of
    // this iteration
}
```

Can be used in lieu of barrier action if parties need not be suspended when run
Overview of Java CyclicBarrier

- Methods wait until all parties call `await()` on barrier & then reset it
- Block until all parties arrive, barrier resets, or thread interrupted
- Block until all parties arrive, barrier resets, thread is interrupted, or timeout elapses

```java
public class CyclicBarrier {
    ...
    public int await() { ... }

    public int await(long timeout, TimeUnit unit) {
        ... }

    public void reset() { ... }
    ...
```
Overview of Java CyclicBarrier

- Methods wait until all parties call await() on barrier & then reset it
  - Block until all parties arrive, barrier resets, or thread interrupted
  - Block until all parties arrive, barrier resets, thread is interrupted, or timeout elapses
- Reset the barrier to the initial state

```java
public class CyclicBarrier {
  ...
  public int await() { ... }
  public int await(long timeout, TimeUnit unit) {
    ... }
  public void reset() { ... }
  ...
}
```

*If any parties are waiting at the barrier, they will return via a BrokenBarrierException*
End of Java Barrier Synchronizers: CyclicBarrier (Part 1)
Java Barrier Synchronizers: CyclicBarrier (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 CyclicBarrier synchronizer allows a set of threads to all wait for each other to reach a common barrier point
- Know how to program with Java CyclicBarrier in practice

```java
class GCDGyclicBarrierTester implements Runnable {
    private final CyclicBarrier mEntryBarrier;
    private final CyclicBarrier mExitBarrier;
    ...

    GCDGyclicBarrierTester(CyclicBarrier entryBarrier,
                           CyclicBarrier exitBarrier, ...) {
        mEntryBarrier = entryBarrier; mExitBarrier = exitBarrier;
    }

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

• An app that uses CyclicBarriers 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/douglasraighschmidt/POSA/tree/master/ex/M3/GCD/CyclicBarrier
Using the Java CyclicBarrier in Practice

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

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

        CyclicBarrier entryBarrier =
            new CyclicBarrier(gcdTests.size() + 1, () ->
                GCDCyclicBarrierTester.initializeInput(sITERATIONS));
        CyclicBarrier exitBarrier =
            new CyclicBarrier(gcdTests.size() + 1);

        for (int cycle = 1; cycle <= sCYCLES; cycle++) {
            gcdTests.forEach(gcdTuple -> new Thread(new
                GCDCyclicBarrierTester(entryBarrier, exitBarrier,
                gcdTuple, this)).start());

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

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

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

        CyclicBarrier entryBarrier =
            new CyclicBarrier(gcdTests.size() + 1, () ->
                GCDCyclicBarrierTester.initializeInput(sITERATIONS));
        CyclicBarrier exitBarrier =
            new CyclicBarrier(gcdTests.size() + 1);

        for (int cycle = 1; cycle <= sCYCLES; cycle++) {
            gcdTests.forEach(gcdTuple -> new Thread(new
                GCDCyclicBarrierTester(entryBarrier, exitBarrier,
                gcdTuple, this)).start());

            System.out.println("Starting tests");
            entryBarrier.await();
            System.out.println("Waiting for results");
            exitBarrier.await();
            System.out.println("All tests done"); ...
        }
    }
}
class GCDCyclicBarrierTest {
    void main() throws InterruptedException {
        List<GCDTuple> gcdTests = makeTests();

        CyclicBarrier entryBarrier =
            new CyclicBarrier(gcdTests.size() + 1, () ->
            GCDCyclicBarrierTester.initializeInput(sITERATIONS));

        CyclicBarrier exitBarrier =
            new CyclicBarrier(gcdTests.size() + 1);

        for (int cycle = 1; cycle <= sCYCLES; cycle++) {
            gcdTests.forEach(gcdTuple -> new Thread(new
                GCDCyclicBarrierTester(entryBarrier, exitBarrier,
                gcdTuple, this)).start());

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

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

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

        CyclicBarrier entryBarrier =
                new CyclicBarrier(gcdTests.size() + 1, () ->
                        GCDCyclicBarrierTester.initializeInput(sITERATIONS));
        CyclicBarrier exitBarrier =
                new CyclicBarrier(gcdTests.size() + 1);

        for (int cycle = 1; cycle <= sCYCLES; cycle++) {
            gcdTests.forEach(gcdTuple ->
                    new Thread(new GCDCyclicBarrierTester(entryBarrier, exitBarrier,
                            gcdTuple, this)).start());
        }

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

Iterate through each cycle
class GCDCyclicBarrierTest {
    void main() throws InterruptedException {
        List<GCDTuple> gcdTests = makeTests();

        CyclicBarrier entryBarrier =
            new CyclicBarrier(gcdTests.size() + 1, () ->
                GCDCyclicBarrierTester.initializeInput(sITERATIONS));
        CyclicBarrier exitBarrier =
            new CyclicBarrier(gcdTests.size() + 1);

        for (int cycle = 1; cycle <= sCYCLES; cycle++) {
            gcdTests.forEach(gcdTuple -> new Thread(new
                GCDCyclicBarrierTester(entryBarrier, exitBarrier,
                gcdTuple, this)).start());

            System.out.println("Starting tests");
            entryBarrier.await();
            System.out.println("Waiting for results");
            exitBarrier.await();
            System.out.println("All tests done"); ...
        }
    }
}
```java
class GCDCyclicBarrierTest {
    void main() throws InterruptedException {
        List<GCDTuple> gcdTests = makeTests();

        CyclicBarrier entryBarrier =
                new CyclicBarrier(gcdTests.size() + 1, () ->
                        GCDCyclicBarrierTester.initializeInput(sITERATIONS));
        CyclicBarrier exitBarrier =
                new CyclicBarrier(gcdTests.size() + 1);

        for (int cycle = 1; cycle <= sCYCLES; cycle++) {
            gcdTests.forEach(gcdTuple ->
                new Thread(new GCDCyclicBarrierTester(entryBarrier, exitBarrier,
                        gcdTuple, this)).start());

            System.out.println("Starting tests"); // Don't start yet
            entryBarrier.await();
            System.out.println("Waiting for results");
            exitBarrier.await();
            System.out.println("All tests done"); ...
        }
    }
}
```

Using the Java CyclicBarrier in Practice

- Create worker threads that use two CyclicBarriers as entry & exit barriers
class GCDCyclicBarrierTest {
    void main() throws InterruptedException {
        List<GCDTuple> gcdTests = makeTests();

        CyclicBarrier entryBarrier =
            new CyclicBarrier(gcdTests.size() + 1, () ->
                GCDCyclicBarrierTester.initializeInput(sITERATIONS));

        CyclicBarrier exitBarrier =
            new CyclicBarrier(gcdTests.size() + 1);

        for (int cycle = 1; cycle <= sCYCLES; cycle++) {
            gcdTests.forEach(gcdTuple -> new Thread(new
                GCDCyclicBarrierTester(entryBarrier, exitBarrier,
                gcdTuple, this)).start());

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

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

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

        CyclicBarrier entryBarrier =
            new CyclicBarrier(gcdTests.size() + 1, () ->
                GCDCyclicBarrierTester.initializeInput(sITERATIONS));
        CyclicBarrier exitBarrier =
            new CyclicBarrier(gcdTests.size() + 1);

        for (int cycle = 1; cycle <= sCYCLES; cycle++) {
            gcdTests.forEach(gcdTuple -> new Thread(new
                GCDCyclicBarrierTester(entryBarrier, exitBarrier,
                gcdTuple, this)).start());

            System.out.println("Starting tests");
            entryBarrier.await();
            System.out.println("Waiting for results");
            exitBarrier.await() \[\textbf{Exit barrier waits for all threads to finish this cycle}\]
            System.out.println("All tests done"); ...
    }
}
```
Using the Java CyclicBarrier in Practice

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

Define a worker that runs in a thread

class GCDCyclicBarrierTester implements Runnable {
    private final CyclicBarrier mEntryBarrier;
    private final CyclicBarrier mExitBarrier;
    ...

    GCDCyclicBarrierTester(CyclicBarrier entryBarrier,
                           CyclicBarrier exitBarrier, ...) {
        mEntryBarrier = entryBarrier; mExitBarrier = exitBarrier;
        ...
    }

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

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

```java
class GCDCyclicBarrierTester implements Runnable {
    private final CyclicBarrier mEntryBarrier;
    private final CyclicBarrier mExitBarrier;
    ...

    GCDCyclicBarrierTester(CyclicBarrier entryBarrier, CyclicBarrier exitBarrier, ...) {
        mEntryBarrier = entryBarrier; mExitBarrier = exitBarrier;
        ...
    }

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

Initialize barrier fields
Using the Java CyclicBarrier in Practice

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

```java
class GCDCyclicBarrierTester implements Runnable {
    private final CyclicBarrier mEntryBarrier;
    private final CyclicBarrier mExitBarrier;
    ...

    GCDCyclicBarrierTester(CyclicBarrier entryBarrier,
                           CyclicBarrier exitBarrier, ...) {
        mEntryBarrier = entryBarrier; mExitBarrier = exitBarrier;
        ...
    }

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

Wait until all threads are ready to run (entry barrier)
Using the Java CyclicBarrier in Practice

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

class GCDCyclicBarrierTester implements Runnable {
    private final CyclicBarrier mEntryBarrier;
    private final CyclicBarrier mExitBarrier;
    ...

    GCDCyclicBarrierTester(CyclicBarrier entryBarrier,
                           CyclicBarrier exitBarrier, ...) {
        mEntryBarrier = entryBarrier; mExitBarrier = exitBarrier;
        ...
    }

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

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

```java
class GCDCyclicBarrierTester implements Runnable {
    private final CyclicBarrier mEntryBarrier;
    private final CyclicBarrier mExitBarrier;
    ...

    GCDCyclicBarrierTester(CyclicBarrier entryBarrier,
                           CyclicBarrier exitBarrier, ...) {
        mEntryBarrier = entryBarrier; mExitBarrier = exitBarrier;
    ...

    public void run() {
        ...
        mEntryBarrier.await();
        runTest();
        mExitBarrier.await(); // Wait until all threads are done
        ...
    }
}
```
End of Java Barrier Synchronizers: CyclicBarrier (Part 2)