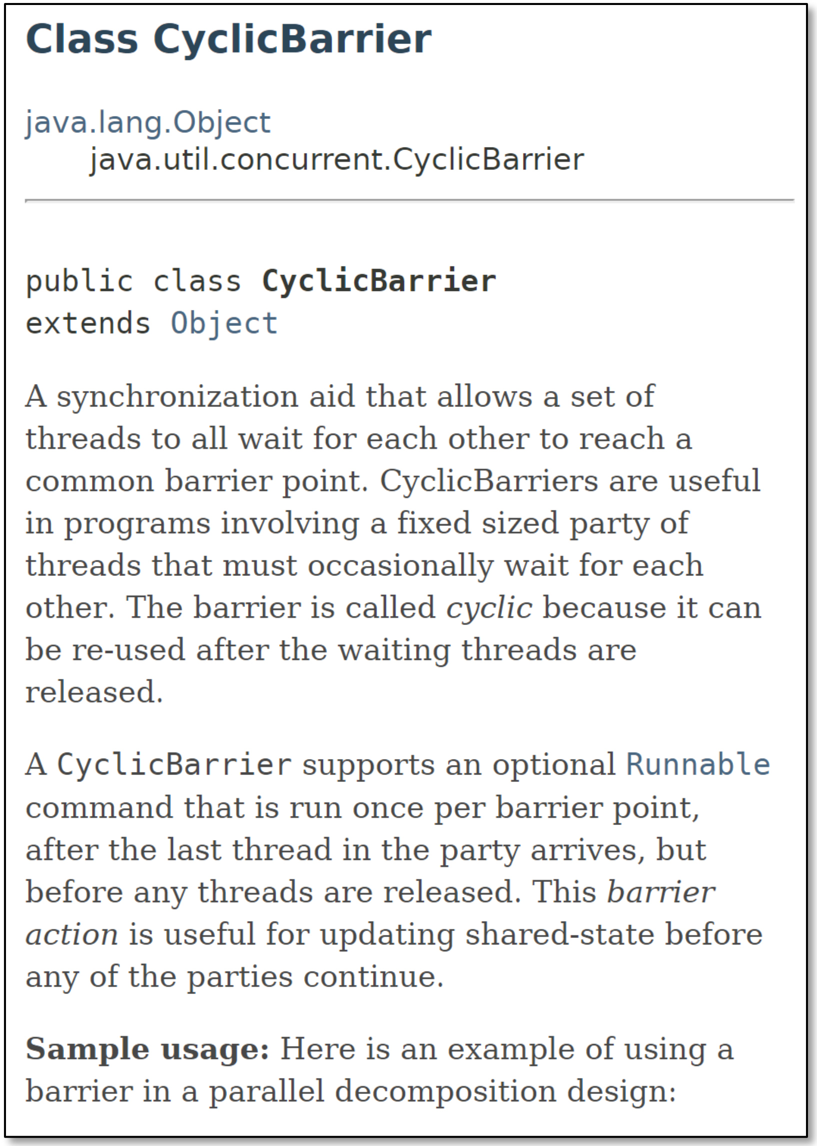
Structure & Functionality of Java CyclicBarrier

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 the structure & functionality of Java 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.

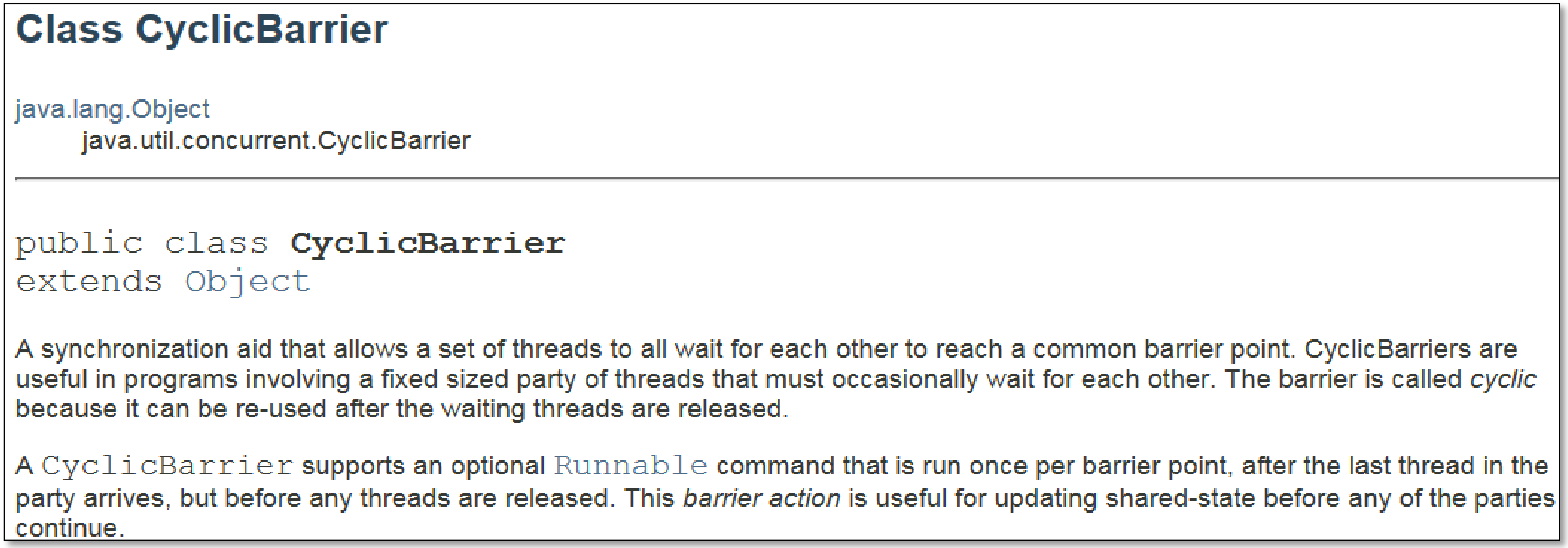
Sample usage: Here is an example of using a barrier in a parallel decomposition design:

Overview of Java CyclicBarrier

Overview of Java CyclicBarrier

- Implements another Java barrier synchronizer

```
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.

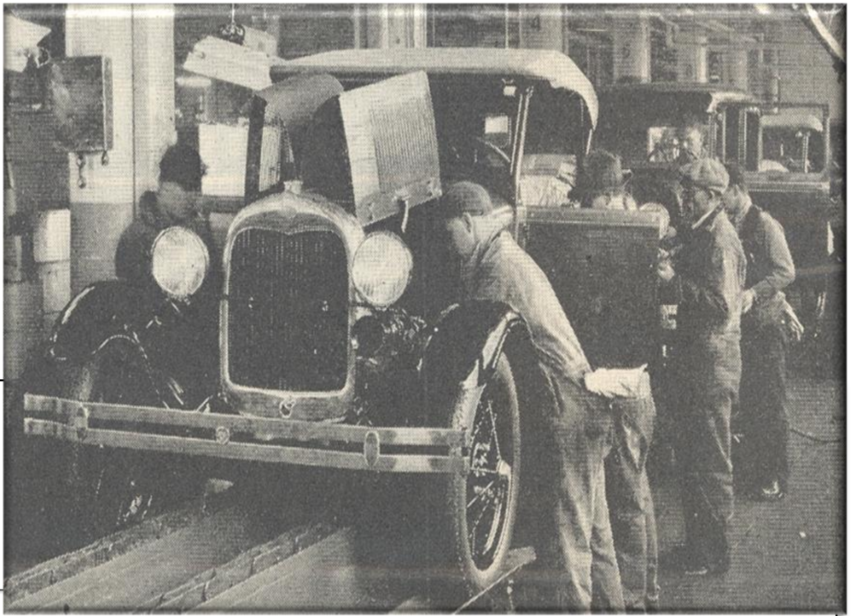
A `CyclicBarrier` supports an optional `Runnable` command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This *barrier action* is useful for updating shared-state before any of the parties continue.

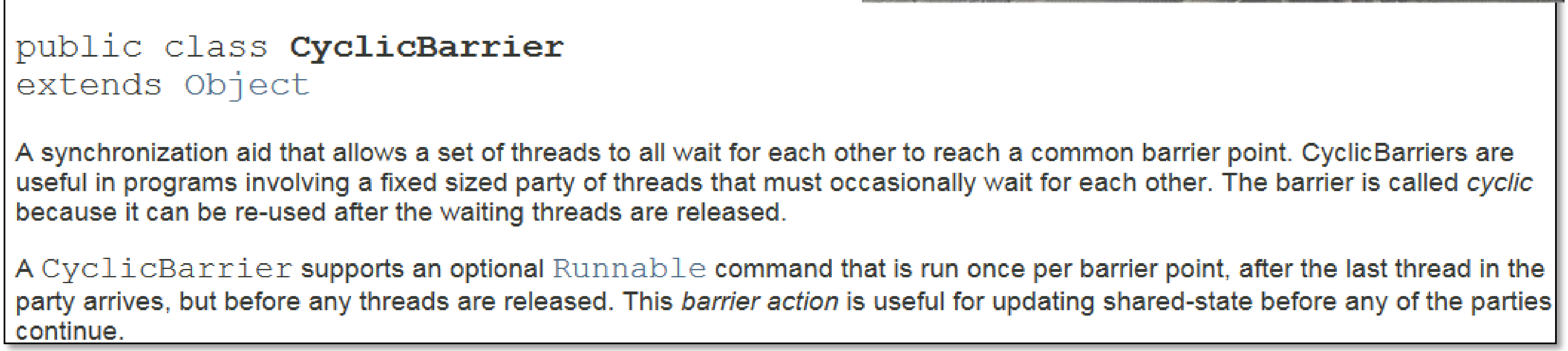
See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html

Overview of Java CyclicBarrier

- Implements another Java barrier synchronizer
  - Allows a set of threads to wait for each other to reach a common barrier point
    - Threads are referred to as “parties”

```
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.

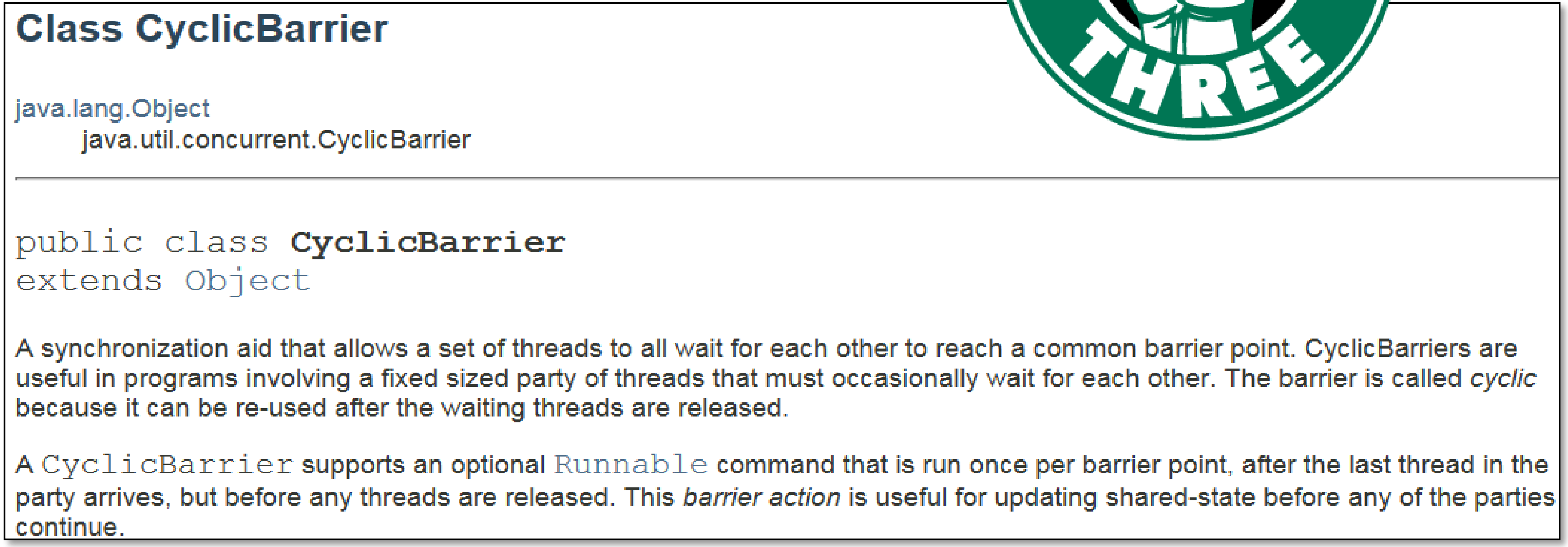
A `CyclicBarrier` supports an optional `Runnable` command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This *barrier action* is useful for updating shared-state before any of the parties continue.

One human known use is an assembly line where fixed-sized groups of workers coordinate to build various parts of cars moving by in phases

Overview of Java CyclicBarrier

- Implements another Java barrier synchronizer
  - Allows a set of threads to wait for each other to reach a common barrier point
  - Well-suited for fixed-size "cyclic", "entry", and/or "exit" barriers

```
public class CyclicBarrier {
  ...
```



Class CyclicBarrier

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

```
public class CyclicBarrier
extends Object
```

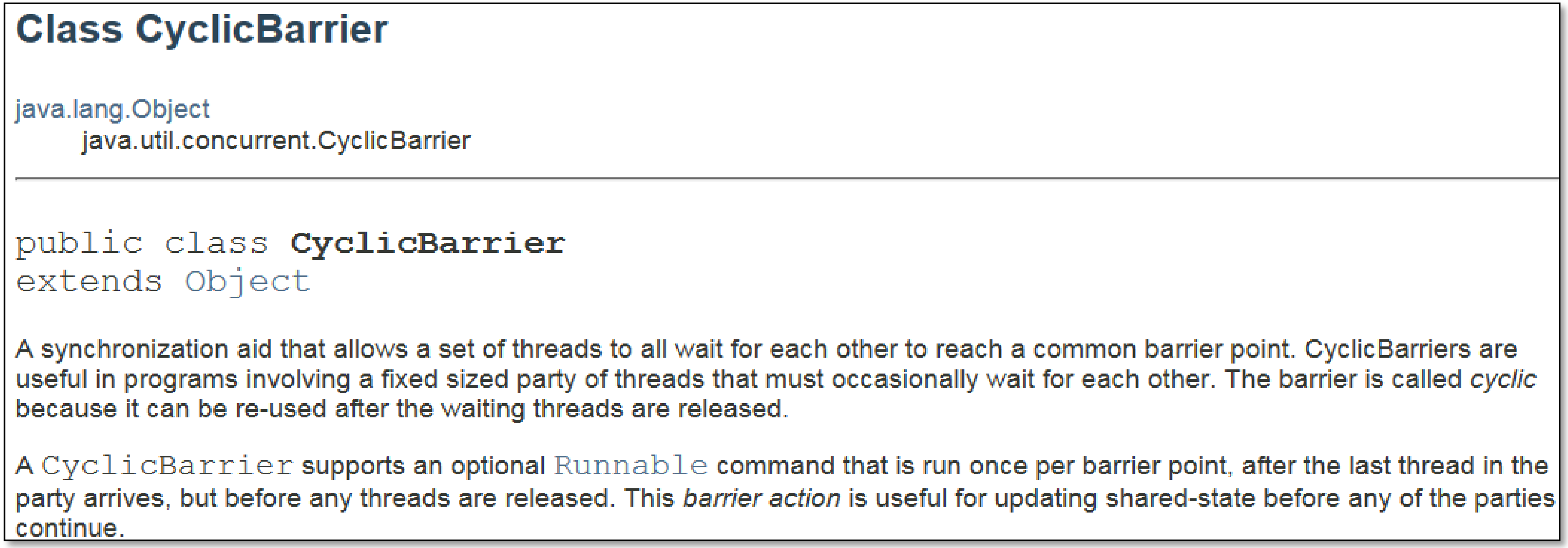
A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called *cyclic* because it can be re-used after the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

Overview of Java CyclicBarrier

- Implements another Java barrier synchronizer
  - Allows a set of threads to wait for each other to reach a common barrier point
  - Well-suited for fixed-size "cyclic", "entry", and/or "exit" barriers
  - Enables barrier to be reset manually at any point

```
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.

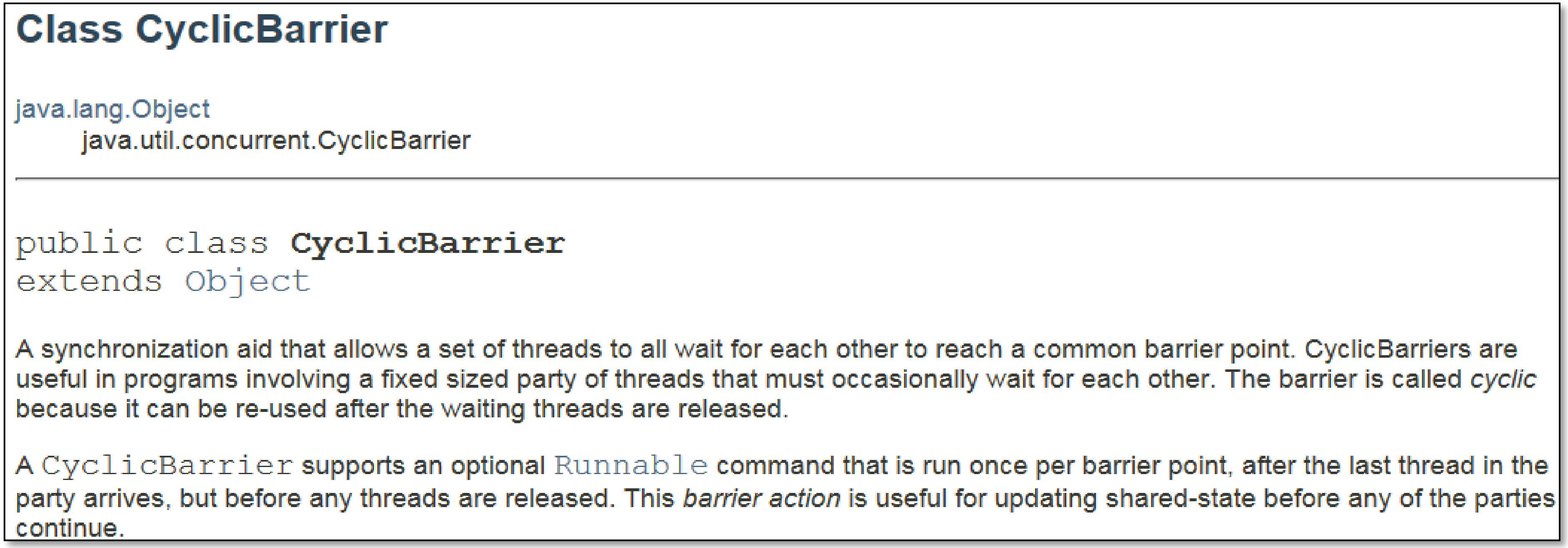
In contrast, Java CountDownLatch does *not* enable the barrier to be reset!

Overview of Java CyclicBarrier

- Implements another Java barrier synchronizer
  - Allows a set of threads to wait for each other to reach a common barrier point
  - Well-suited for fixed-size "cyclic", "entry", and/or "exit" barriers
  - Enables barrier to be reset manually at any point

```
public class CyclicBarrier {
  ...
```

Does not implement an interface



Class CyclicBarrier

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

```
public class CyclicBarrier
extends Object
```

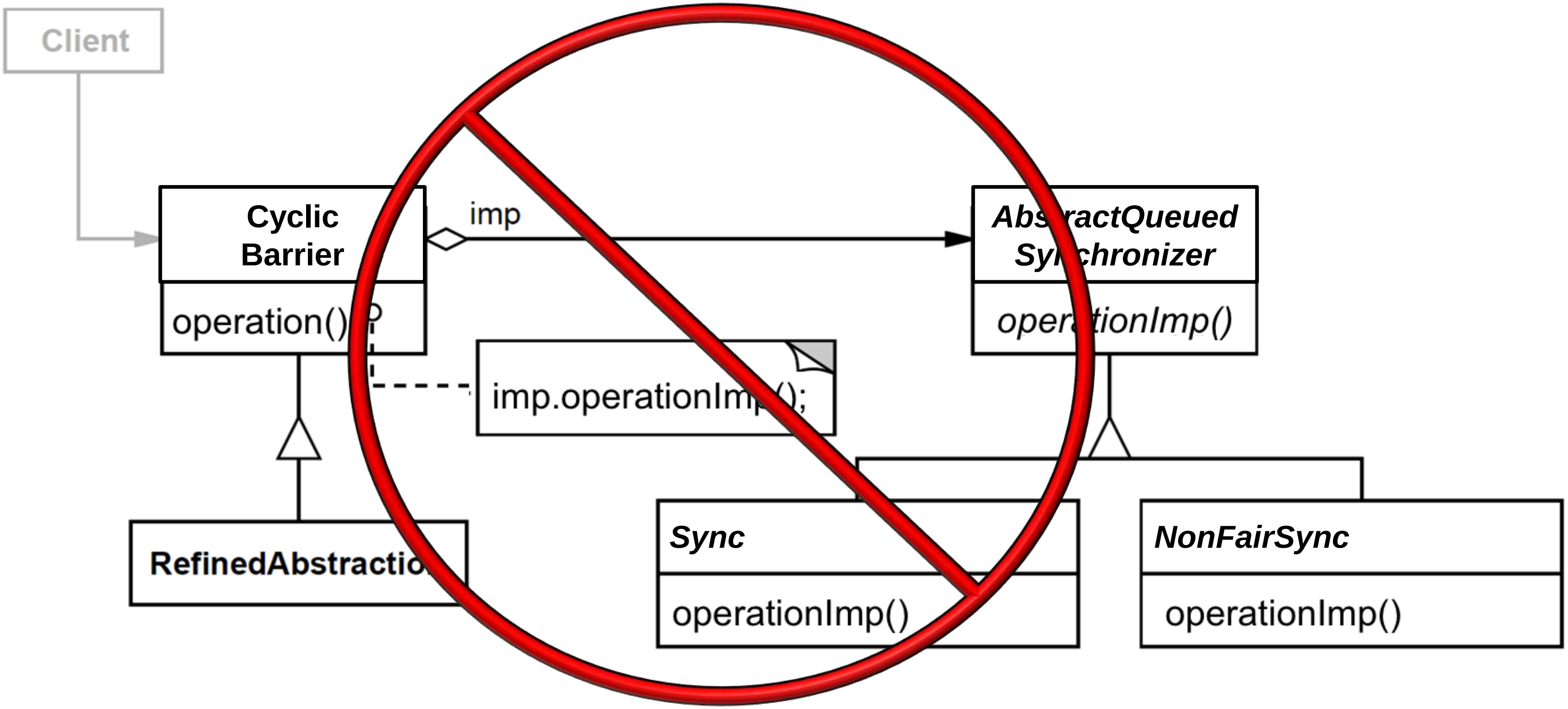
A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called *cyclic* because it can be re-used after the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

Overview of Java CyclicBarrier

- Does not apply the *Bridge* pattern

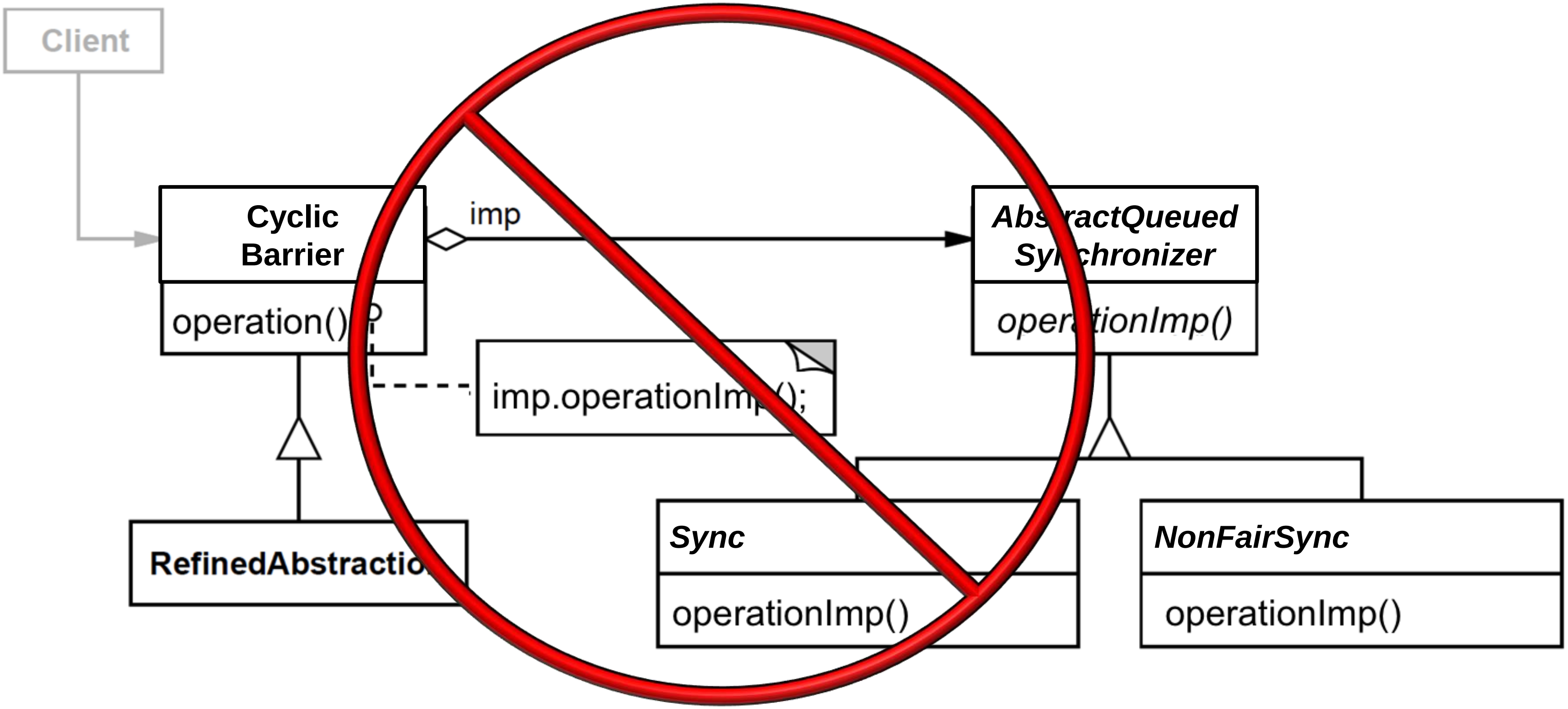
```
public class CyclicBarrier {
  ...
```



Overview of Java CyclicBarrier

- Does not apply the *Bridge* pattern
  - Nor does it use the Abstract QueuedSynchronizer framework

```
public class CyclicBarrier {
  ...
```



Unlike the Java ReentrantLock, ReentrantReadWriteLock, Semaphore, ConditionObject, & CountDownLatch classes

Overview of Java CyclicBarrier

- Instead, it defines a # of fields that implement a cyclic barrier

```
public class CyclicBarrier {
  private final ReentrantLock
    lock = new ReentrantLock();

  private final Condition trip
    = lock.newCondition();

  private final int parties;

  private int count;

  private final Runnable
    barrierCommand;

  ...
```

See src/share/classes/java/util/concurrent/CyclicBarrier.java

Overview of Java CyclicBarrier

- Instead, it defines a # of fields that implement a cyclic barrier
  - Lock that protects critical sections

```
public class CyclicBarrier {
  private final ReentrantLock
    lock = new ReentrantLock();

  private final Condition trip
    = lock.newCondition();

  private final int parties;

  private int count;

  private final Runnable
    barrierCommand;

  ...
```

Overview of Java CyclicBarrier

- Instead, it defines a # of fields that implement a cyclic barrier
  - Lock that protects critical sections
  - Condition to wait on until tripped

```
public class CyclicBarrier {
  private final ReentrantLock
    lock = new ReentrantLock();

  private final Condition trip
    = lock.newCondition();

  private final int parties;

  private int count;

  private final Runnable
    barrierCommand;

  ...
```

Overview of Java CyclicBarrier

- Instead, it defines a # of fields that implement a cyclic barrier
  - Lock that protects critical sections
  - Condition to wait on until tripped
  - The total # of parties
    - This value is initially set by the CyclicBarrier constructor

```
public class CyclicBarrier {
  private final ReentrantLock
    lock = new ReentrantLock();

  private final Condition trip
    = lock.newCondition();

  private final int parties;

  private int count;

  private final Runnable
    barrierCommand;

  ...
```

Overview of Java CyclicBarrier

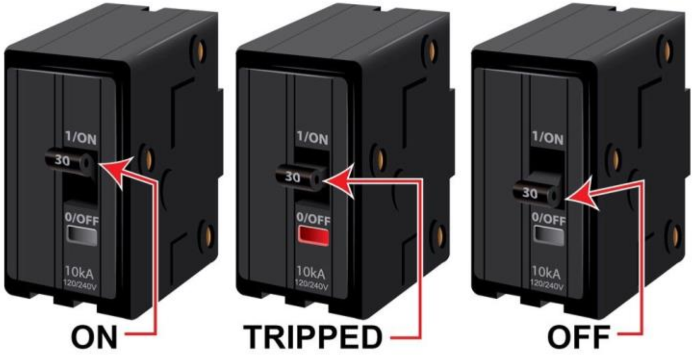
- Instead, it defines a # of fields that implement a cyclic barrier
  - Lock that protects critical sections
  - Condition to wait on until tripped
  - The total # of parties
  - # of parties that haven't called await() yet
    - Initially set to total # of parties & then decremented each time await() is called

```
public class CyclicBarrier {
  private final ReentrantLock
    lock = new ReentrantLock();

  private final Condition trip
    = lock.newCondition();

  private final int parties;

  private int count;

  private final Runnable
    barrierCommand;

  ...
```

Overview of Java CyclicBarrier

- Instead, it defines a # of fields that implement a cyclic barrier
  - Lock that protects critical sections
  - Condition to wait on until tripped
  - The total # of parties
  - # of parties that haven't called await() yet
  - Barrier action (optional)
    - Called when barrier is "tripped" after all parties arrive

```
public class CyclicBarrier {
  private final ReentrantLock
    lock = new ReentrantLock();

  private final Condition trip
    = lock.newCondition();

  private final int parties;

  private int count;

  private final Runnable
    barrierCommand;

  ...
```



Overview of Java CyclicBarrier

- Instead, it defines a # of fields that implement a cyclic barrier
  - Lock that protects critical sections
  - Condition to wait on until tripped
  - The total # of parties
  - # of parties that haven’t called await() yet
  - Barrier action (optional)
    - Called when barrier is “tripped” after all parties arrive
      - An action is typically used to (re)initialize data structures for the next cycle

```
public class CyclicBarrier {
  private final ReentrantLock
    lock = new ReentrantLock();

  private final Condition trip
    = lock.newCondition();

  private final int parties;

  private int count;

  private final Runnable
    barrierCommand;

  ...
```

End of Structure & Functionality of Java CyclicBarrier