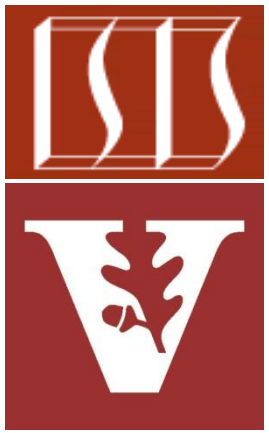
Java Phaser:
Structure & Functionality



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 the Java Phaser barrier synchronizer

Class Phaser

java.lang.Object
    java.util.concurrent.Phaser

```
public class Phaser
extends Object
```

A reusable synchronization barrier, similar in functionality to `CyclicBarrier` and `CountDownLatch` but supporting more flexible usage.

Registration. Unlike the case for other barriers, the number of parties *registered* to synchronize on a phaser may vary over time. Tasks may be registered at any time (using methods `register()`, `bulkRegister(int)`, or forms of constructors establishing initial numbers of parties), and optionally deregistered upon any arrival (using `arriveAndDeregister()`). As is the case with most basic synchronization constructs, registration and deregistration affect only internal counts; they do not establish any further internal bookkeeping, so tasks cannot query whether they are registered. (However, you can introduce such bookkeeping by subclassing this class.)

Overview of Java Phaser

Overview of Java Phaser

- Implements yet another Java barrier synchronizer

```
public class Phaser {
  ...
```

Class Phaser

java.lang.Object
    java.util.concurrent.Phaser

```
public class Phaser
extends Object
```

A reusable synchronization barrier, similar in functionality to `CyclicBarrier` and `CountDownLatch` but supporting more flexible usage.

Registration. Unlike the case for other barriers, the number of parties *registered* to synchronize on a phaser may vary over time. Tasks may be registered at any time (using methods `register()`, `bulkRegister(int)`, or forms of constructors establishing initial numbers of parties), and optionally deregistered upon any arrival (using `arriveAndDeregister()`). As is the case with most basic synchronization constructs, registration and deregistration affect only internal counts; they do not establish any further internal bookkeeping, so tasks cannot query whether they are registered. (However, you can introduce such bookkeeping by subclassing this class.)

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

Overview of Java Phaser

- Implements yet another Java barrier synchronizer
  - Allows a variable (or fixed) # of threads to wait for all operations performed in other threads to complete before proceeding

```
public class Phaser {
  ...
```

Class Phaser

java.lang.Object
    java.util.concurrent.Phaser

```
public class Phaser
extends Object
```

A reusable synchronization barrier, similar in functionality to `CyclicBarrier` and `CountDownLatch` but supporting more flexible usage.

Registration. Unlike the case for other barriers, the number of parties *registered* to synchronize on a phaser may vary over time. Tasks may be registered at any time (using methods `register()`, `bulkRegister(int)`, or forms of constructors establishing initial numbers of parties), and optionally deregistered upon any arrival (using `arriveAndDeregister()`). As is the case with most basic synchronization constructs, registration and deregistration affect only internal counts; they do not establish any further internal bookkeeping, so tasks cannot query whether they are registered. (However, you can introduce such bookkeeping by subclassing this class.)

One human known use is different work-crews with different #'s of workers coordinating to build a house

Overview of Java Phaser

- Implements yet another Java barrier synchronizer
  - Allows a variable (or fixed) # of threads to wait for all operations performed in other threads to complete before proceeding
  - Well-suited for variable-size "cyclic", "entry", and/or "exit" barriers

```
public class Phaser {
  ...
```

Class Phaser

java.lang.Object
    java.util.concurrent.Phaser

```
public class Phaser
extends Object
```

A reusable synchronization barrier, similar in functionality to `CyclicBarrier` and `CountDownLatch` but supporting more flexible usage.

Registration. Unlike the case for other barriers, the number of parties *registered* to synchronize on a phaser may vary over time. Tasks may be registered at any time (using methods `register()`, `bulkRegister(int)`, or forms of constructors establishing initial numbers of parties), and optionally deregistered upon any arrival (using `arriveAndDeregister()`). As is the case with most basic synchronization constructs, registration and deregistration affect only internal counts; they do not establish any further internal bookkeeping, so tasks cannot query whether they are registered. (However, you can introduce such bookkeeping by subclassing this class.)

Overview of Java Phaser

- Implements yet another Java barrier synchronizer
  - Allows a variable (or fixed) # of threads to wait for all operations performed in other threads to complete before proceeding
  - Well-suited for variable-size "cyclic", "entry", and/or "exit" barriers
  - # of parties can vary dynamically

```
public class Phaser {
  ...
```

Class Phaser

java.lang.Object
    java.util.concurrent.Phaser

```
public class Phaser
extends Object
```

A reusable synchronization barrier, similar in functionality to `CyclicBarrier` and `CountDownLatch` but supporting more flexible usage.

Registration. Unlike the case for other barriers, the number of parties *registered* to synchronize on a phaser may vary over time. Tasks may be registered at any time (using methods `register()`, `bulkRegister(int)`, or forms of constructors establishing initial numbers of parties), and optionally deregistered upon any arrival (using `arriveAndDeregister()`). As is the case with most basic synchronization constructs, registration and deregistration affect only internal counts; they do not establish any further internal bookkeeping, so tasks cannot query whether they are registered. (However, you can introduce such bookkeeping by subclassing this class.)

A Phaser may be overkill for fixed-sized barriers..

Overview of Java Phaser

- Implements yet another Java barrier synchronizer
  - Allows a variable (or fixed) # of threads to wait for all operations performed in other threads to complete before proceeding
  - Well-suited for variable-size "cyclic", "entry", and/or "exit" barriers
  - # of parties can vary dynamically

```
public class Phaser {
  ...
```

Does not implement an interface

Class Phaser

java.lang.Object
    java.util.concurrent.Phaser

```
public class Phaser
extends Object
```

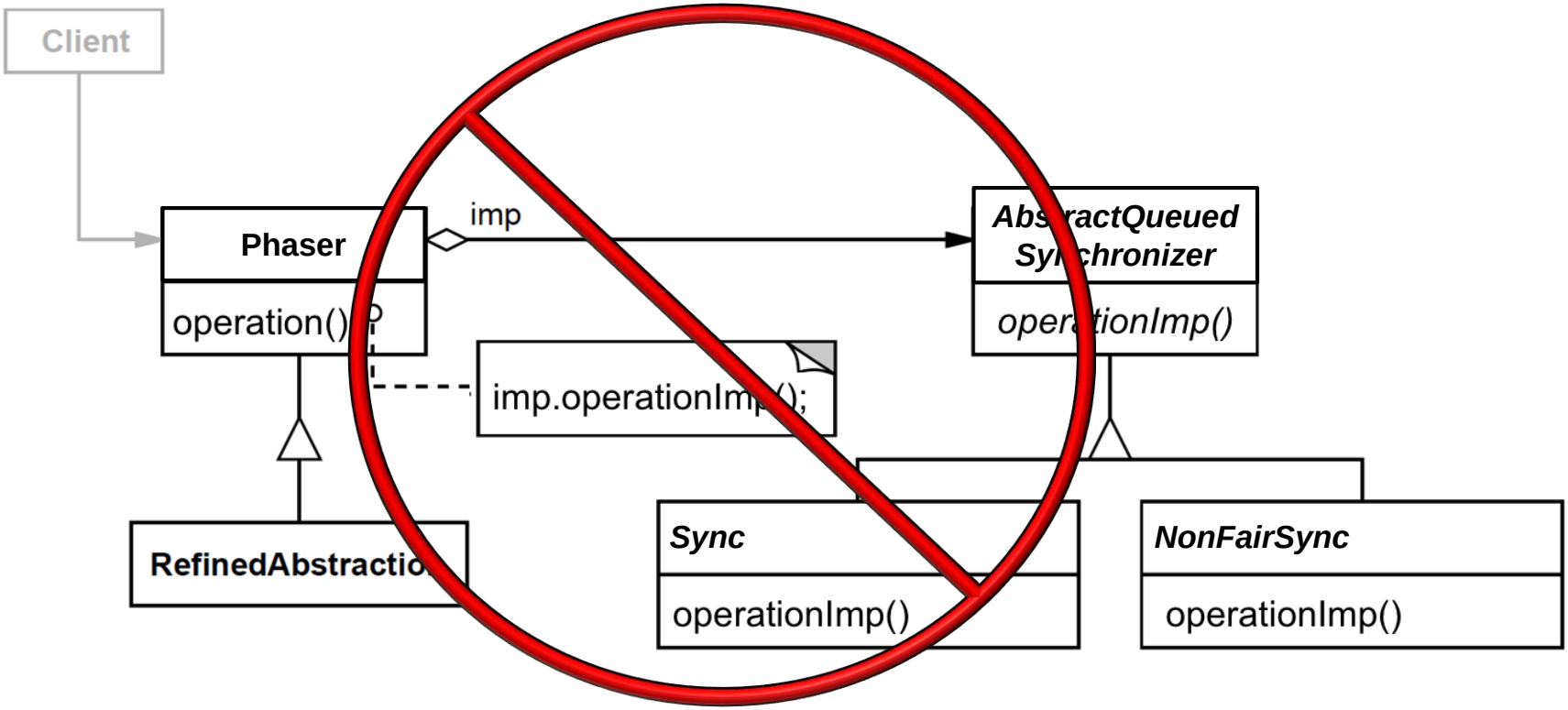
A reusable synchronization barrier, similar in functionality to `CyclicBarrier` and `CountDownLatch` but supporting more flexible usage.

Registration. Unlike the case for other barriers, the number of parties *registered* to synchronize on a phaser may vary over time. Tasks may be registered at any time (using methods `register()`, `bulkRegister(int)`, or forms of constructors establishing initial numbers of parties), and optionally deregistered upon any arrival (using `arriveAndDeregister()`). As is the case with most basic synchronization constructs, registration and deregistration affect only internal counts; they do not establish any further internal bookkeeping, so tasks cannot query whether they are registered. (However, you can introduce such bookkeeping by subclassing this class.)

Overview of Java Phaser

- Does not apply *Bridge* pattern

```
public class Phaser {
  ...
```

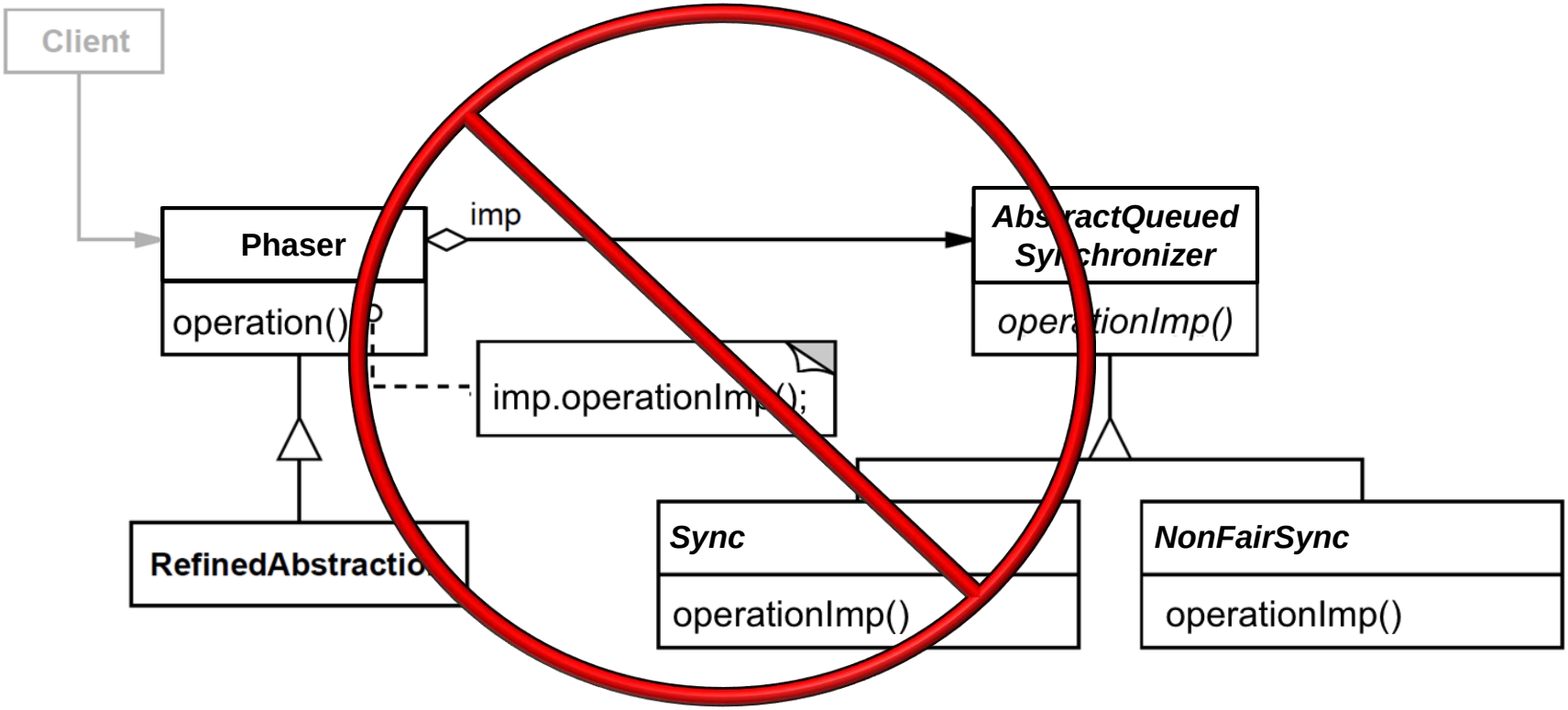


See share/classes/java/util/concurrent/Phaser.java

Overview of Java Phaser

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

```
public class Phaser {
   ...
```



Overview of Java Phaser

- Instead, it defines a # of fields that implement a phaser

```
public class Phaser {
  private volatile long state;
  ...
```

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

Overview of Java Phaser

- Instead, it defines a # of fields that implement a phaser
  - Primary state representation, holding four bit-fields

```
public class Phaser {
  private volatile long state;
```

See en.wikipedia.org/wiki/Bit_field

Overview of Java Phaser

- Instead, it defines a # of fields that implement a phaser
  - Primary state representation, holding four bit-fields:
    - *Unarrived*
      - the # of parties yet to hit barrier (bits 0-15)

```
public class Phaser {
  private volatile long state;
```

Overview of Java Phaser

- Instead, it defines a # of fields that implement a phaser
  - Primary state representation, holding four bit-fields:
    - *Unarrived*
    - *Parties*
      - the # of parties to wait for before advancing to the next phase (bits 16-31)

```
public class Phaser {
  private volatile long state;
```

Overview of Java Phaser

- Instead, it defines a # of fields that implement a phaser
  - Primary state representation, holding four bit-fields:
    - *Unarrived*
    - *Parties*
    - *Phase*
      - the generation of the barrier (bits 32-62)

```
public class Phaser {
  private volatile long state;
```

Overview of Java Phaser

- Instead, it defines a # of fields that implement a phaser
  - Primary state representation, holding four bit-fields:
    - *Unarrived*
    - *Parties*
    - *Phase*
    - *Terminated*
      - set if barrier is terminated (bit 63 / sign)

```
public class Phaser {
  private volatile long state;
```

Overview of Java Phaser

- Instead, it defines a # of fields that implement a phaser
  - Primary state representation, holding four bit-fields:
    - *Unarrived*
      - the # of parties yet to hit barrier (bits 0-15)
    - *Parties*
      - the # of parties to wait (bits 16-31)
    - *Phase*
      - the generation of the barrier (bits 32-62)
    - *Terminated*
      - set if barrier is terminated (bit 63 / sign)

```
public class Phaser {
  private volatile long state;
```

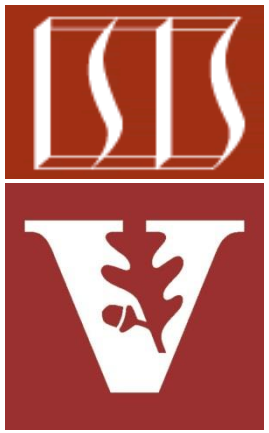
To efficiently maintain atomicity, these values are packed into a single (atomic) long that is updated via CAS operations

End of Java Phaser: Structure & Functionality

Discussion Questions

1. What of the following are benefit of the Java Phaser over the CyclicBarrier?
   a. *It supports fixed-size "cyclic" & "entry" and/or "exit" barriers who # of parties match the # of threads*
   b. *It supports variable-size "cyclic" & "entry" and/or "exit" barriers whose # of parties can vary dynamically*
   c. *It uses the AbstractQueuedSynchronizer framework to enhance reuse*
   d. *They provide better support for fixed-sized # of parties*

Java Phaser:
Key Methods

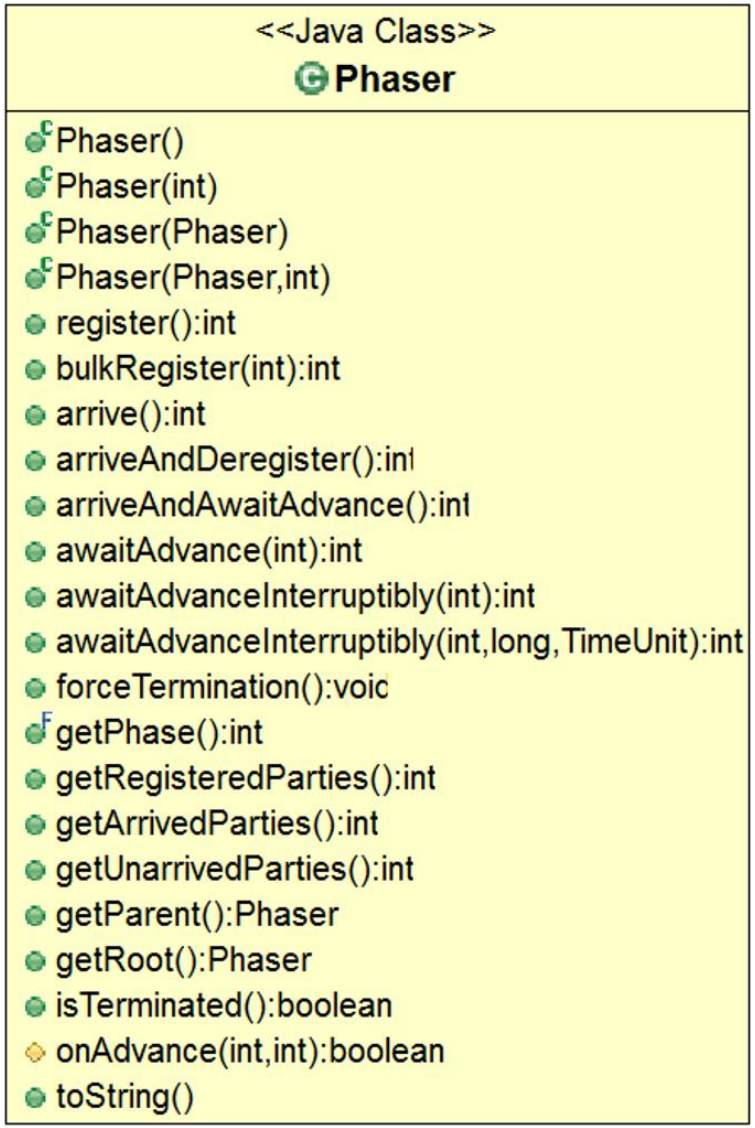


Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of the Java Phaser barrier synchronizer
- Recognize the key methods in the Java Phaser



Key Methods in Java Phaser

Key Methods in Java Phaser

- Constructor creates a new object with an initial phase # of 0

```
public class Phaser {
   ...
   public Phaser() { ... }

   public Phaser(int parties) {
       ...
   }
```

Key Methods in Java Phaser

- Constructor creates a new object with an initial phase # of 0
  - Any thread using this Phaser needs to register for it first

```
public class Phaser {
   ...
   public Phaser() { ... }

   public Phaser(int parties) {
       ...
   }
```

Key Methods in Java Phaser

- Constructor creates a new object with an initial phase # of 0
  - Any thread using this Phaser needs to register for it first
  - Can also specify the # of parties needed to advance to next phase

```
public class Phaser {
   ...
   public Phaser() { ... }

   public Phaser(int parties) {
      ...
   }
```

Key Methods in Java Phaser

- Constructor creates a new object with an initial phase # of 0
  - Any thread using this Phaser needs to register for it first
  - Can also specify the # of parties needed to advance to next phase
    - However, using this constructor is optional since parties can always register later

```
public class Phaser {
   ...
   public Phaser() { ... }

   public Phaser(int parties) {
      ...
   }
```

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate

```
public class Phaser {
  ...
  public int register() { ... }

  public int bulkRegister
    (int parties) { ... }

  public int
    arriveAndAwaitAdvance()
    { ... }

  public int ArriveAndDeregister()
  { ... }

  protected boolean onAdvance
    (int phase,
     int registeredParties) {
    return registeredParties == 0;
  }
```

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser

```
public class Phaser {
   ...
   public int register() { ... }

   public int bulkRegister
      (int parties) { ... }
```

# of registered parties dictates when a phaser can advance

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser
  - Arrive & await advance

```
public class Phaser {
   ...
   public int arrive() { ... }

   public int awaitAdvance
                  (int phase)
   { ... }

   public int
     arriveAndAwaitAdvance()
   { ... }
```

These methods provide flexibility wrt arrival & waiting to advance

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser
  - Arrive & await advance
    - Arrives at phaser without waiting for others to arrive

```
public class Phaser {
  ...
  public int arrive() { ... }

  public int awaitAdvance
                (int phase)
  { ... }

  public int
    arriveAndAwaitAdvance()
  { ... }
```

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser
  - Arrive & await advance
    - Arrives at phaser without waiting for others to arrive
      - Returns arrival phase #

```
public class Phaser {
  ...
  public int arrive() { ... }

  public int awaitAdvance
                (int phase)
  { ... }

  public int
    arriveAndAwaitAdvance()
  { ... }
```

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser
  - Arrive & await advance
    - Arrives at phaser without waiting for others to arrive
      - Returns arrival phase #
        - Or negative value if phaser terminated

```
public class Phaser {
  ...
  public int arrive() { ... }

  public int awaitAdvance
                (int phase)
  { ... }

  public int
    arriveAndAwaitAdvance()
  { ... }
```

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser
  - Arrive & await advance
    - Arrives at phaser without waiting for others to arrive
    - Awaits the phase of this phaser to advance from the given phase value

```
public class Phaser {
  ...
  public int arrive() { ... }

  public int awaitAdvance
                (int phase)
  { ... }

  public int
    arriveAndAwaitAdvance()
  { ... }
```

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser
  - Arrive & await advance
    - Arrives at phaser without waiting for others to arrive
    - Awaits the phase of this phaser to advance from the given phase value
      - Returns immediately if current phrase != given phase

```
public class Phaser {
  ...
  public int arrive() { ... }

  public int awaitAdvance
                (int phase)
  { ... }

  public int
    arriveAndAwaitAdvance()
  { ... }
```

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser
  - Arrive & await advance
    - Arrives at phaser without waiting for others to arrive
    - Awaits the phase of this phaser to advance from the given phase value
    - Arrives at phaser & awaits the arrival of others

```
public class Phaser {
  ...
  public int arrive() { ... }

  public int awaitAdvance
                (int phase)
  { ... }

  public int
    arriveAndAwaitAdvance()
  { ... }
```

Equivalent in effect to awaitAdvance(arrive())

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser
  - Arrive & await advance
  - Arrive at the phaser & deregister without waiting for others to arrive

```
public class Phaser {
   ...
   public int arriveAndDeregister()
   { ... }
```

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser
  - Arrive & await advance
  - Arrive at the phaser & deregister without waiting for others to arrive
    - Reduces # of parties required to advance in future phases

```
public class Phaser {
   ...
   public int arriveAndDeregister()
   { ... }
```

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser
  - Arrive & await advance
  - Arrive at the phaser & deregister without waiting for others to arrive
  - Hook method performs an action upon pending phase advance

```
public class Phaser {
   ...
   protected boolean onAdvance
      (int phase,
       int registeredParties) {
      return registeredParties == 0;
   }
```

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser
  - Arrive & await advance
  - Arrive at the phaser & deregister without waiting for others to arrive
  - Hook method performs an action upon pending phase advance
    - Also initiates termination by returning true

```
public class Phaser {
   ...
   protected boolean onAdvance
      (int phase,
       int registeredParties) {
      return registeredParties == 0;
   }
```

Key Methods in Java Phaser

- Phaser's key methods enable parties to register, synchronize, & terminate
  - Adds unarrived parties to phaser
  - Arrive & await advance
  - Arrive at the phaser & deregister without waiting for others to arrive
  - Hook method performs an action upon pending phase advance
    - Also initiates termination by returning true

```
public class Phaser {
   ...
   protected boolean onAdvance
      (int phase,
       int registeredParties) {
     return registeredParties == 0;
   }
```

Default implementation terminates if there are no registered parties

End of Java Phaser: Key Methods

Discussion Questions

1. What of the following are benefit of the Java Phaser over the CyclicBarrier?
   a. *It supports fixed-size "cyclic" & "entry" and/or "exit" barriers who # of parties match the # of threads*
   b. *It supports variable-size "cyclic" & "entry" and/or "exit" barriers whose # of parties can vary dynamically*
   c. *It uses the AbstractQueuedSynchronizer framework to enhance reuse*
   d. *They provide better support for fixed-sized # of parties*