Thread Safety with Phaser

Dr Heinz M. Kabutz

heinz@javaspecialists.eu

Javaspecialists.eu
java training

Phaser

- You will learn
  - What type of problems Phaser aims to solve
  - How it differs from other synchronizers
  - What is "special" about Phaser
- This tutorial assumes a good working knowledge of threading
  - To learn more, be sure to get our Mastering Threads Course
    - learning.javaspecialists.eu
  - Also join our free The Java Specialists' Newsletter
    - javaspecialists.eu/archive

CountDownLatch

- **Blocks until count reaches zero**
  - Once it reaches zero, it remains open forever
- **For example, wait until**
  - All resources have been initialized
  - All services have been started
  - All horses are at the gate

Code Sample: CountDownLatch

```
Service getService() throws InterruptedException {
  serviceCountDown.await();
  return service;
}
```

```
void startDb() {
  db = new Database();
  db.start();
  serviceCountDown.countDown();
}
```

```
void startMailServer() {
  mail = new MailServer();
  mail.start();
  serviceCountDown.countDown();
}
```

Interface: CountDownLatch

```
public class CountDownLatch {
 CountDownLatch(int count)
```

Fixed number of initial "permits"

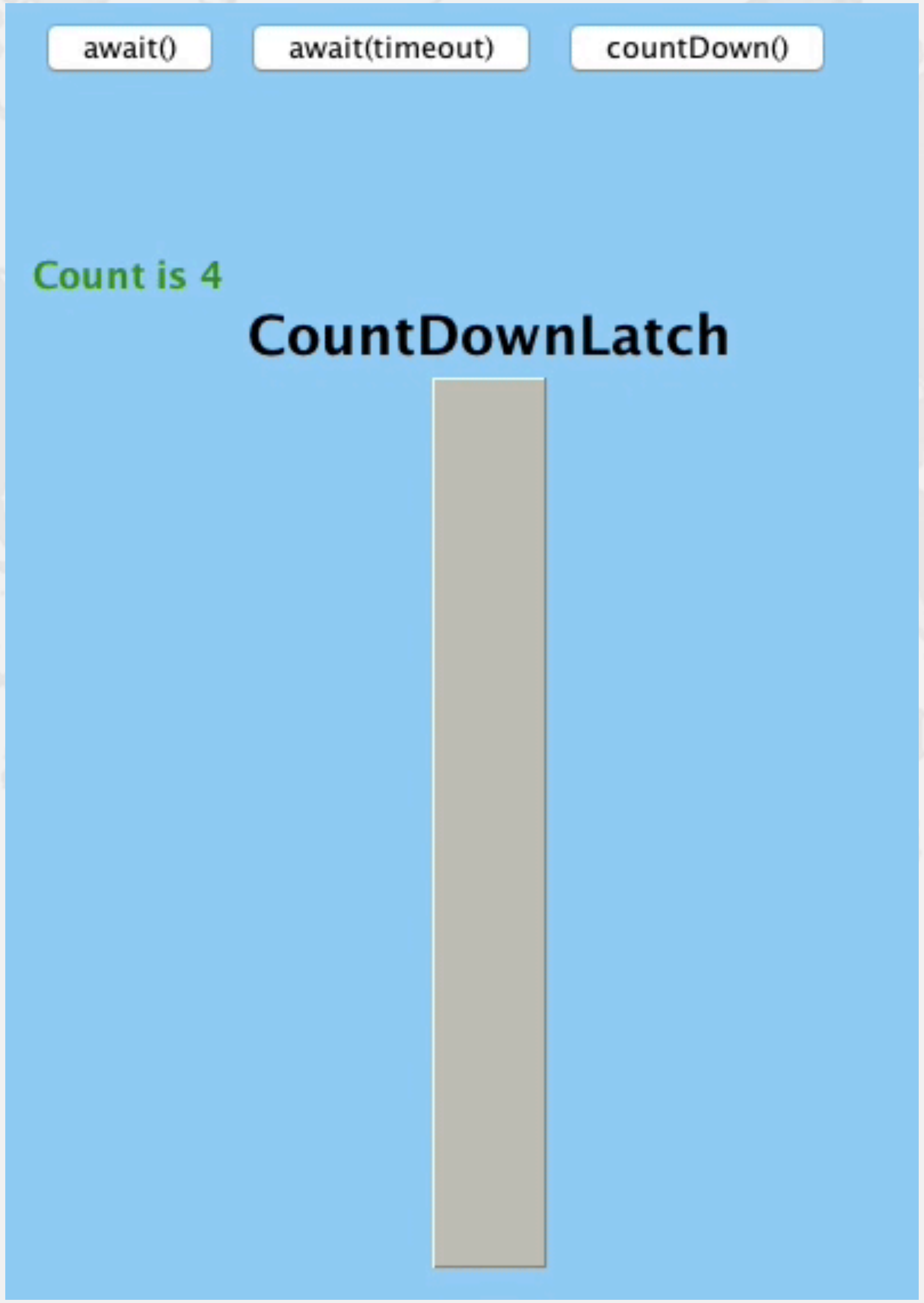
A thread can wait for count to reach zero

```
  void await() throws InterruptedException
  boolean await(long timeout, TimeUnit unit)
                        throws InterruptedException



  void countDown()
}
```

We can count down, but never up. No reset possible.

Animation

- by Victor Grazi
  - – www.jconcurrent.com
- Threads wait until latch is 0



CyclicBarrier

- Similar to CountDownLatch
  - Threads block until all have reached the same point
  - But then it is reset to the initial value
- CyclicBarrier allows a fixed number of parties to rendezvous repeatedly at a barrier point
- Constructor takes an optional "barrier action" Runnable
  - The Runnable is executed when the barrier is successfully passed but before the blocked threads are released.

Interface: CyclicBarrier

Fixed number of parties meet regularly

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

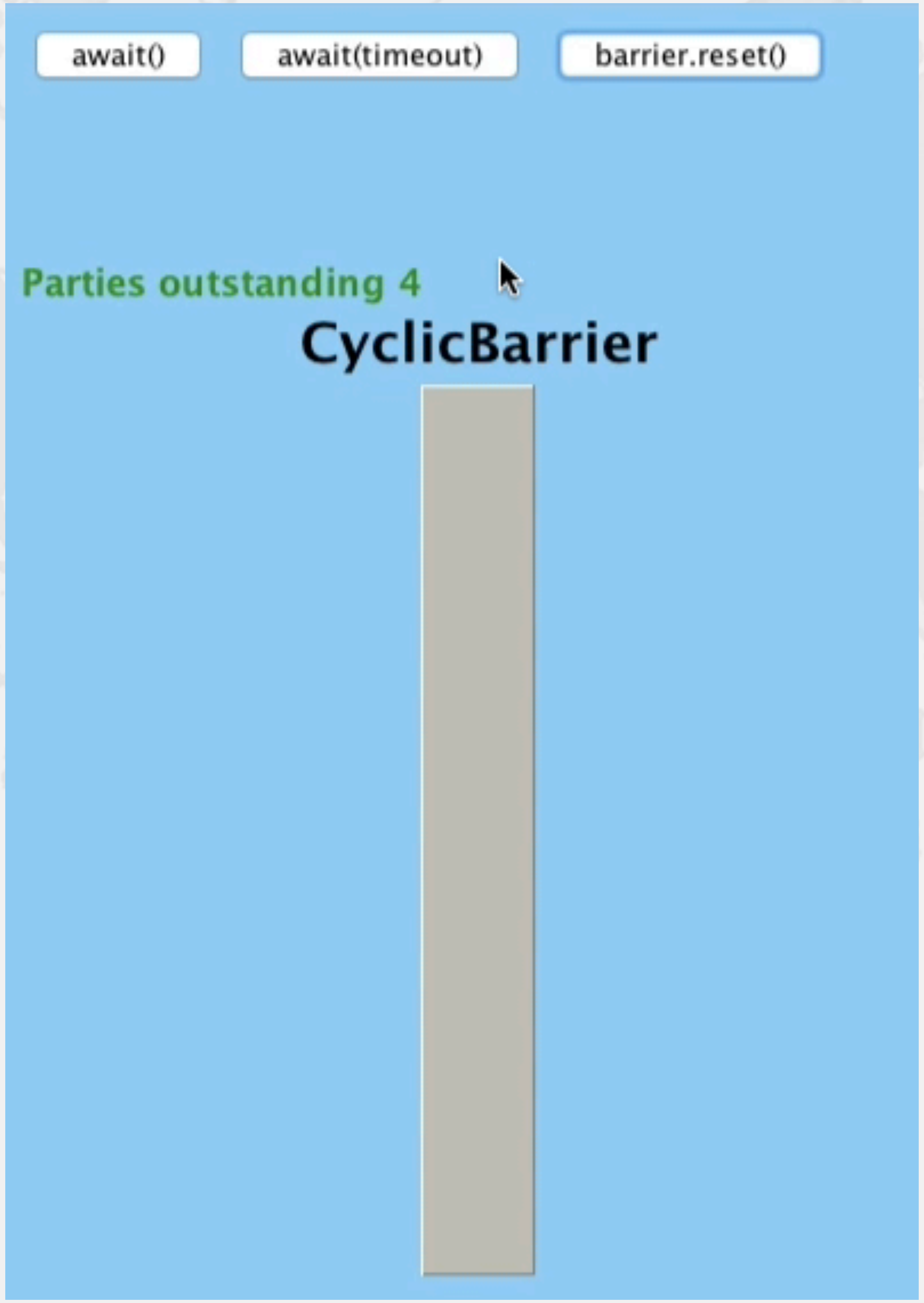
await() waits for all of the threads to arrive

```
  int await() throws InterruptedException,
                     BrokenBarrierException
  int await(long timeout, TimeUnit unit)
              throws InterruptedException,
                     BrokenBarrierException,
                     TimeoutException


  void reset()
}
```

If one of the parties times out, the barrier is broken and must be reset

Animation

- by Victor Grazi
  - www.jconcurrent.com
- Broken barriers need to be reset



Phaser

- **Mix of CyclicBarrier and CountDownLatch**
  - **Number of parties *registered* may vary over time**
    - **Like *count* in CountDownLatch and *parties* in CyclicBarrier**
  - **More modern approach to InterruptedException**
- **Compatible with Fork/Join framework**
  - **Use ManagedBlocker**

Interface: Phaser Registration

```
public class Phaser {
  Phaser(Phaser parent, int parties)
```

Phasers can be arranged in tree to reduce contention

Parameters are optional

```
  int register()

  int bulkRegister(int parties)
```

We can change the parties dynamically by calling register()

Interface: Phaser Signal/Wait

```
public class Phaser {

  int arrive()
  int arriveAndDeregister()
```

Signal only

```
  int awaitAdvance(int phase)
```

Wait only - default is to save interrupt

```
  int awaitAdvanceInterruptibly(int phase[, timeout])
    throws InterruptedException


  int arriveAndAwaitAdvance()
```

Signal and wait - also saves interrupt

Interface: Phaser Action

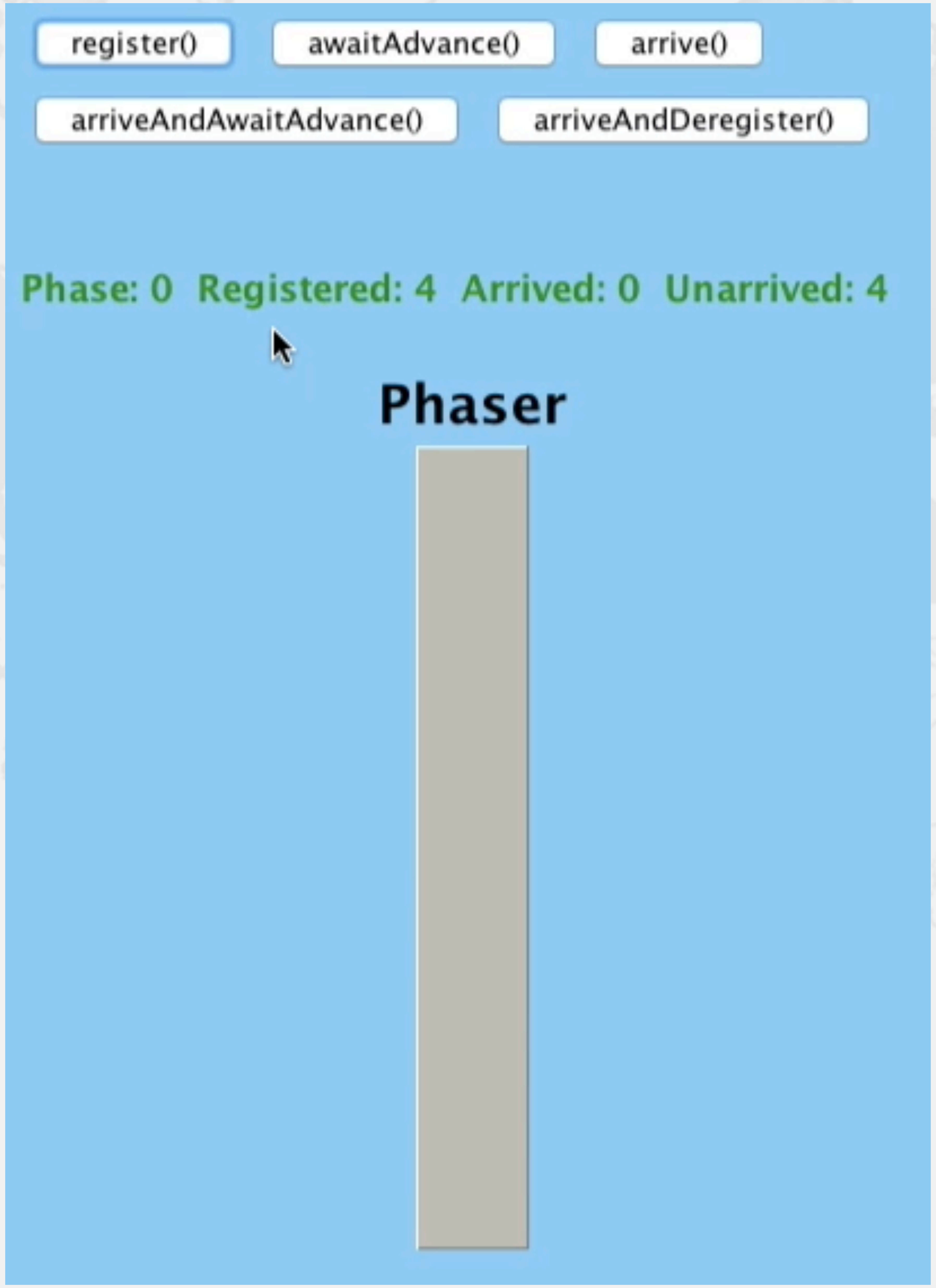
```
public class Phaser {
  protected boolean onAdvance(
    int phase, int registeredParties)

}
```

Override onAdvance() to let phaser finish early

Bunch of lifecycle methods left out

Animation

- by Victor Grazi
  - – www.jconcurrent.com



Phase: 0 Registered: 4 Arrived: 0 Unarrived: 4

Demo: Coordinated Start of Tasks

- Several tasks should start their work together
  - Or as close as possible, subject to OS scheduling
  - Need at least 4 physical cores
  - We will use the Epsilon GC
- We will code different approaches
  - None
  - wait/notify and Lock/Condition/await/signal
  - Volatile and acquire/release spin
  - CountDownLatch and CyclicBarrier
  - Phaser

Counting Phases

- Phaser keeps score of phase we are in
  - CyclicBarrier does not
- We can use this to cancel the Phaser

```
private void addButtons(int buttons, int blinks) {
  Phaser phaser = new Phaser(buttons) {
    protected boolean onAdvance(
        int phase, int registeredParties) {
      return phase >= blinks - 1 ||
             registeredParties == 0;
    }
  };

  // ...
```

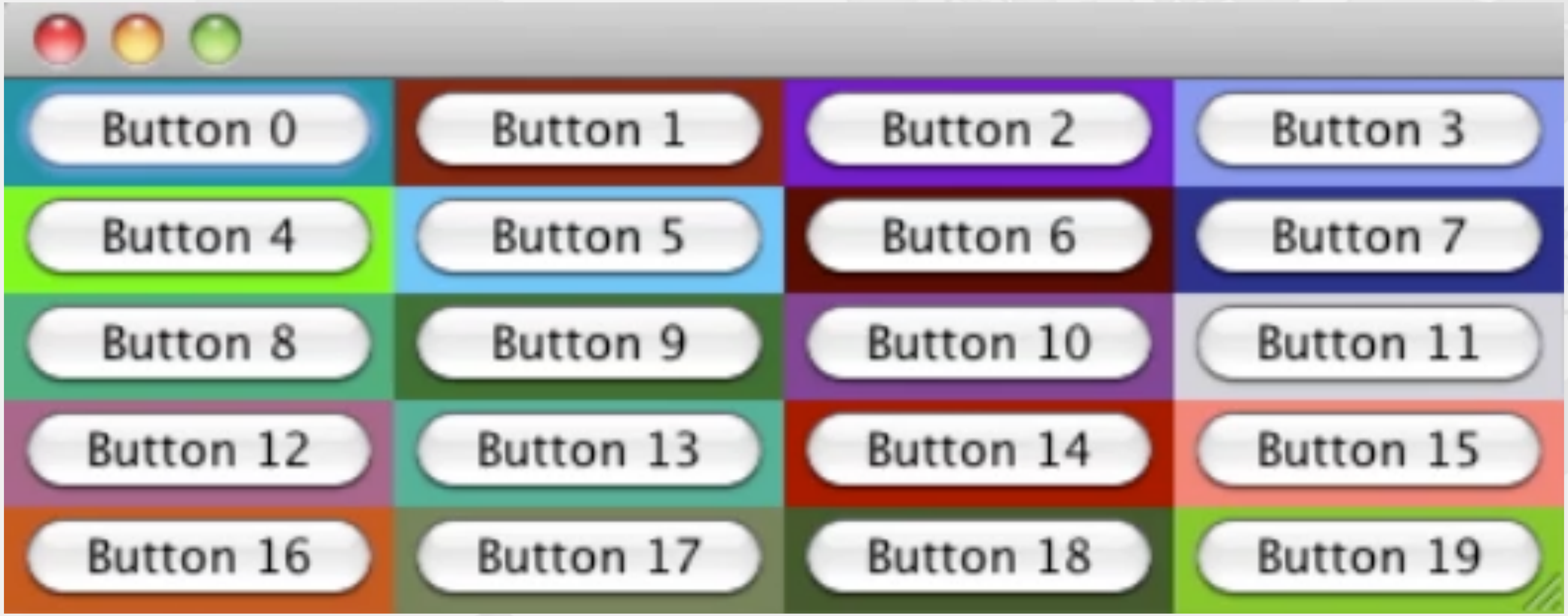
Random Colors on Buttons

- **We change color until Phaser is terminated**

```
new Thread() {
  public void run() {
    Random rand = ThreadLocalRandom.current();
    try {
      do {
        Color newColor = new Color(rand.nextInt());
        changeColor(comp, newColor); // sets it with the EDT
        Thread.sleep(rand.nextInt(500, 3000));
        changeColor(comp, defaultColor);
        Toolkit.getDefaultToolkit().beep();
        Thread.sleep(2000);
        phaser.arriveAndAwaitAdvance();
      } while (!phaser.isTerminated());
    } catch (InterruptedException e) { return; }
  }
}.start();
```

20 Buttons and 3 Phases

- **All phases start at the same time**
  - **And end when the color is reset to original**



Tiered Phasers

- Tree of phasers can reduce contention
- A bit complicated to understand (at least for me)
  - Parent does not know what children it has
  - When a child is added, parent # parties increases by 1
    - If child's registered parties > 0
  - Once child arrived parties == 0, one party automatically arrives at parent
  - With arriveAndAwaitAdvance(), we wait for all parties in tree
    - Thus the parties in the current phaser and in the parent have to arrive

Tiered Phasers

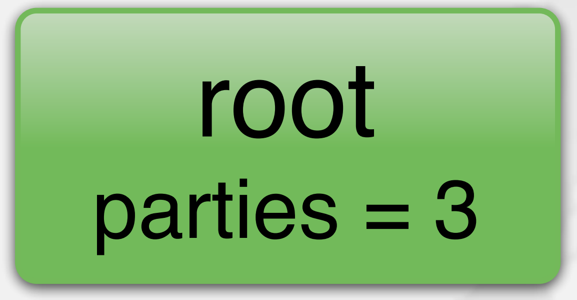
- **Parent parties incremented when child has parties**

```
Phaser root = new Phaser(3);
Phaser c1 = new Phaser(root, 4);
Phaser c2 = new Phaser(root, 5);
Phaser c3 = new Phaser(c2, 0);
System.out.println(c3);
System.out.println(c2);
System.out.println(c1);
System.out.println(root);
```

- **outputs**

```
j.u.c.Phaser[phase = 0 parties = 0 arrived = 0]  (c3)
j.u.c.Phaser[phase = 0 parties = 5 arrived = 0]  (c2)
j.u.c.Phaser[phase = 0 parties = 4 arrived = 0]  (c1)
j.u.c.Phaser[phase = 0 parties = 5 arrived = 0]  (root)
```

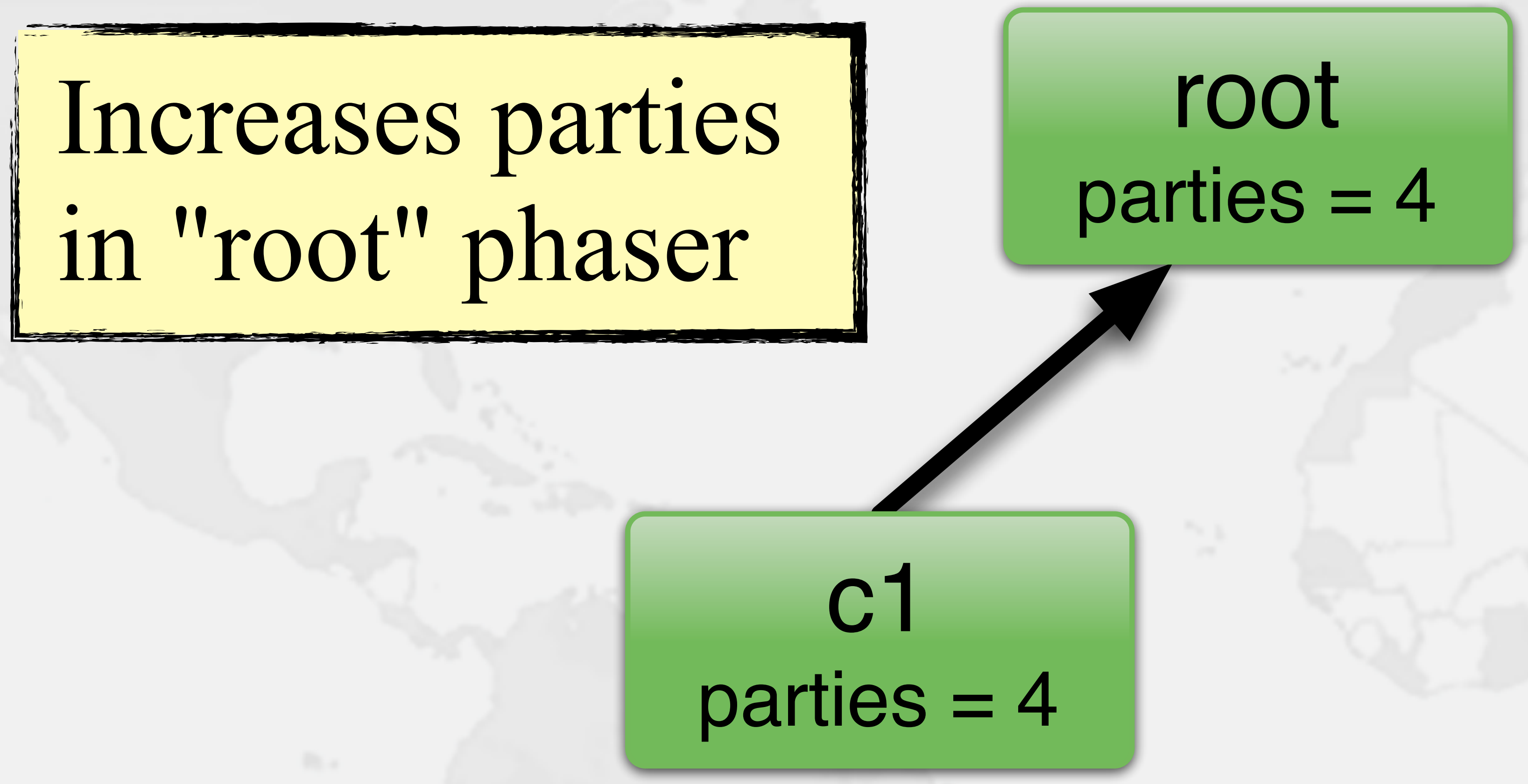
Phaser "root" is Created With 3 Parties



root

parties = 3

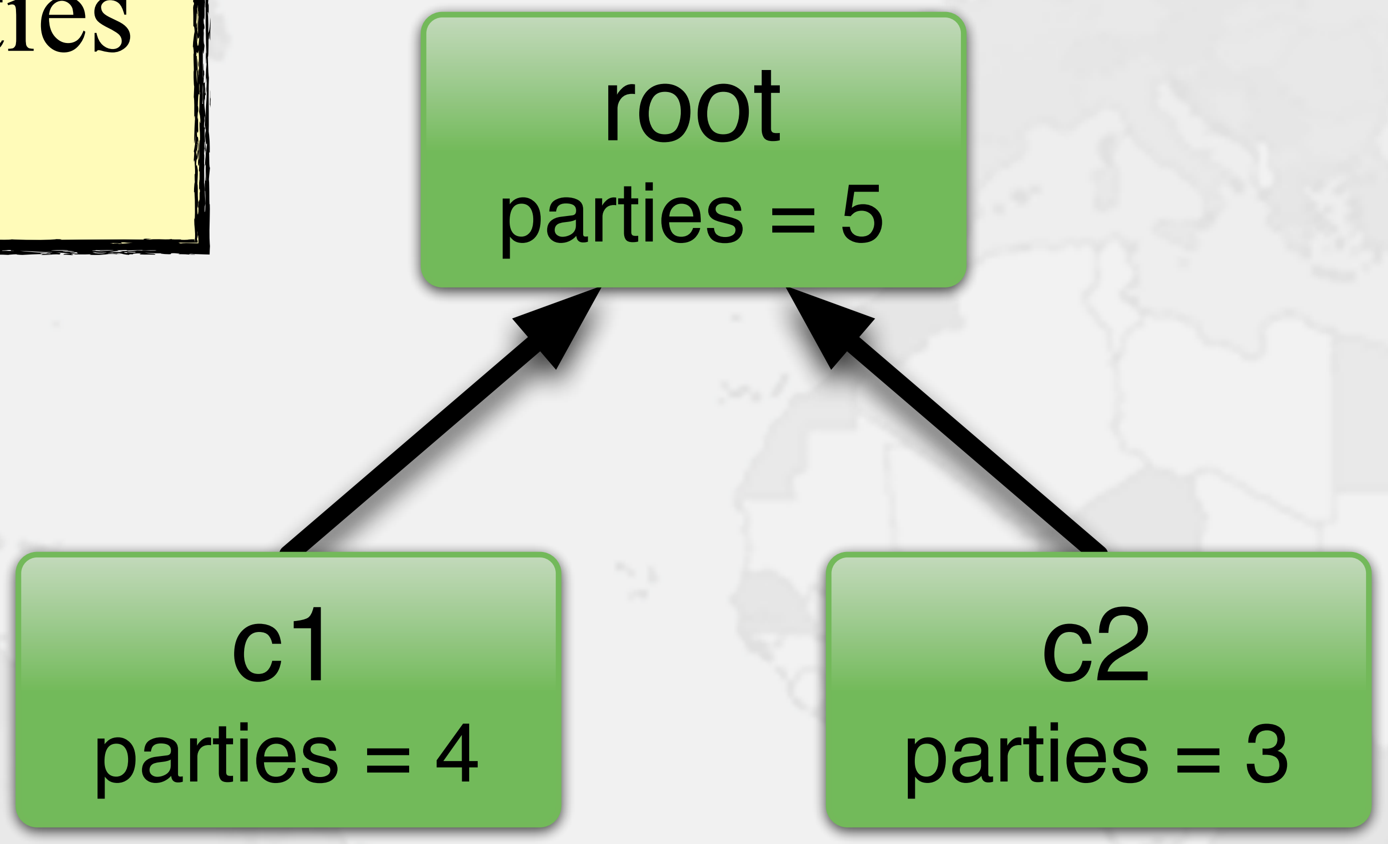
Phaser "c1" is Created With 4 Parties



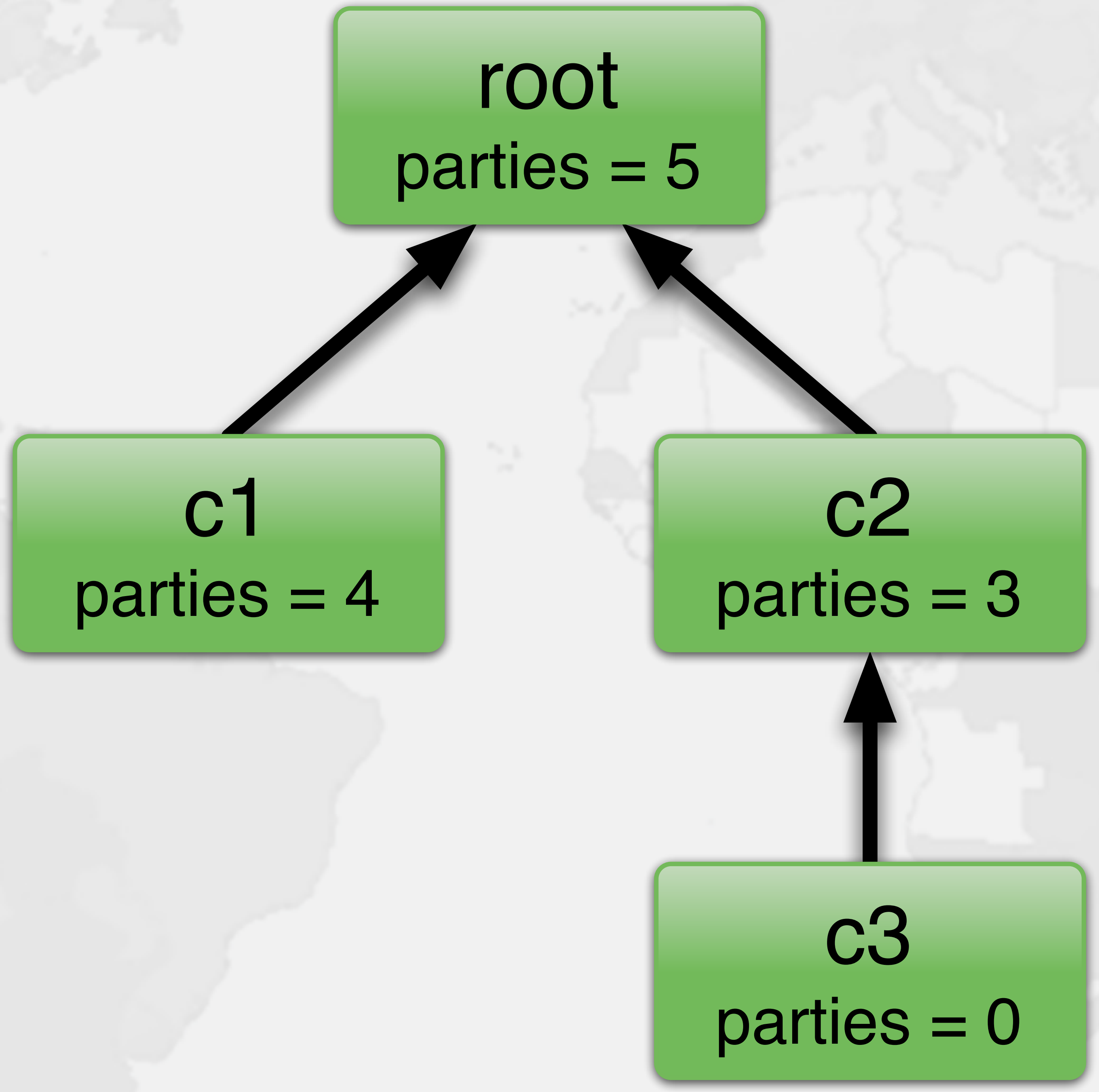
Increases parties in "root" phaser

Phaser "c2" is created with 3 parties

Again increases parties in "root" phaser



Phaser "c3" is created with 0 parties



Does **not** increase parties in "c2" phaser, because c3's parties == 0

ManagedBlocker

- **[JavaDoc] Phasers may also be used by tasks executing in a ForkJoinPool which will ensure sufficient parallelism to execute tasks when others are blocked waiting for a phase to advance.**
- **Fork/Join Pools do not typically have an upper limit on threads**
  - The pool will try have *active threads* equal to desired *parallelism level*
    - Additional threads might be created temporarily

Thread Safety with Phaser

```java
public class ForkJoinPhaser {
  public static void main(String[] args) {
    ForkJoinPool fjp = new ForkJoinPool();
    fjp.invoke(new PhasedAction(100, new Phaser(100)));
    System.out.println(fjp);
  }
  private static class PhasedAction extends RecursiveAction {
    private final int phases;
    private final Phaser ph;
    private PhasedAction(int phases, Phaser ph) {
      this.phases = phases;
      this.ph = ph;
    }
    protected void compute() {
      if (phases == 1) {
        System.out.printf("wait: %s%n", Thread.currentThread());
        ph.arriveAndAwaitAdvance();
        System.out.printf("done: %s%n", Thread.currentThread());
      } else {
        int left = phases / 2;
        int right = phases - left;
        invokeAll(new PhasedAction(left, ph),
                  new PhasedAction(right, ph));
      }
    }
  }
}
```

Additional Threads Maintain Parallelism

```
done: Thread[ForkJoinPool-1-worker-227,5,main]
done: Thread[ForkJoinPool-1-worker-239,5,main]
done: Thread[ForkJoinPool-1-worker-197,5,main]
done: Thread[ForkJoinPool-1-worker-209,5,main]
done: Thread[ForkJoinPool-1-worker-253,5,main]
done: Thread[ForkJoinPool-1-worker-139,5,main]
done: Thread[ForkJoinPool-1-worker-167,5,main]
done: Thread[ForkJoinPool-1-worker-179,5,main]
done: Thread[ForkJoinPool-1-worker-207,5,main]
ForkJoinPool[
  Running,
  parallelism = 12,
  size = 100,
  active = 0, running = 0, steals = 100,
  tasks = 0, submissions = 0]
```

Synchronizers Summary

- CountDownLatch
  - Threads wait for latch to count down to zero
- CyclicBarrier
  - Threads rendezvous at a barrier
- Phaser
  - Flexible synchronizer for task coordination

Further Resources

- The Java Specialists' Newsletter
  - Essential reading for anyone serious about Java
  - `www.javaspecialists.eu`
- Online Bootcamp for Java Specialists
  - 150+ hours of Java lessons
  - `learning.javaspecialists.eu`
- Concurrency Interest Mailing List
  - `g.oswego.edu/dl/concurrency-interest`
- Email: `heinz@javaspecialists.eu`
- Twitter: `@heinzkabutz`