Controlling Concurrent Applications Using Barriers and Latches

José Paumard

PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

Agenda

Two more concurrent primitives

The barrier: to have several tasks wait for each other

The latch: to count down operations and let a task start

Barriers

Stating the Problem

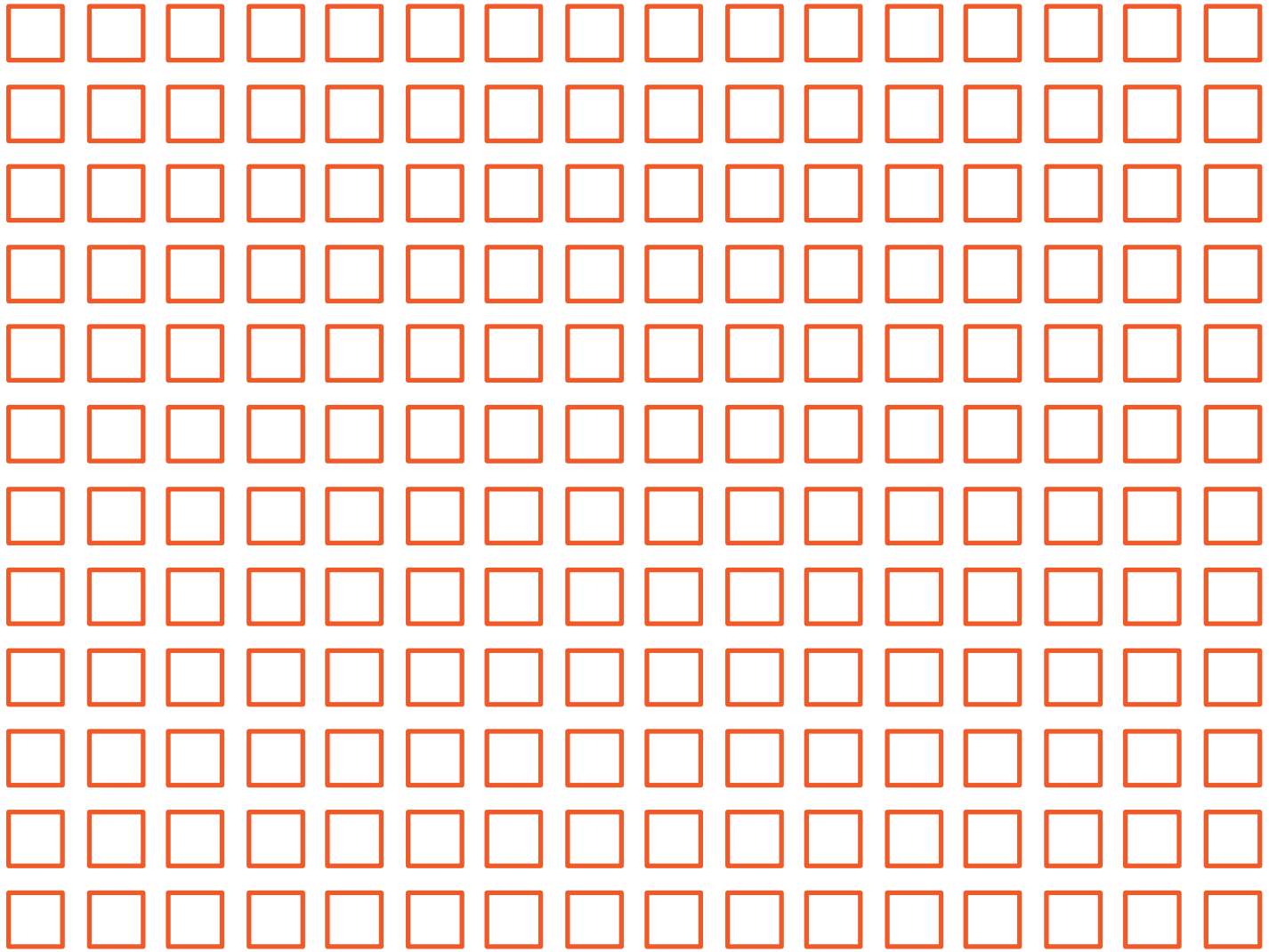
We need a given computation to be shared among several threads

Each thread is given a subtask

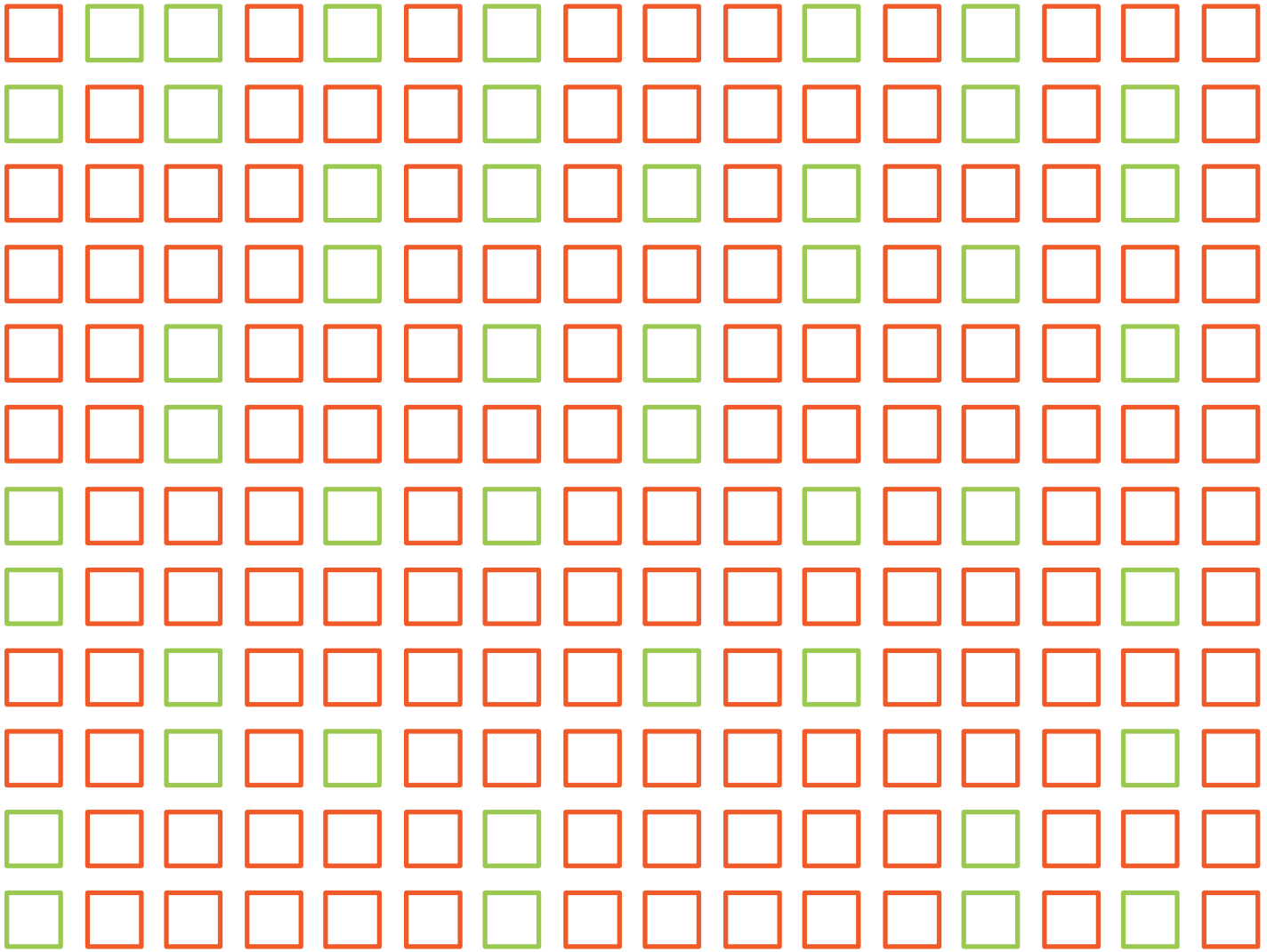
When all the threads are done, then a merging operation is run

Finding Prime Numbers

Finding Prime Numbers



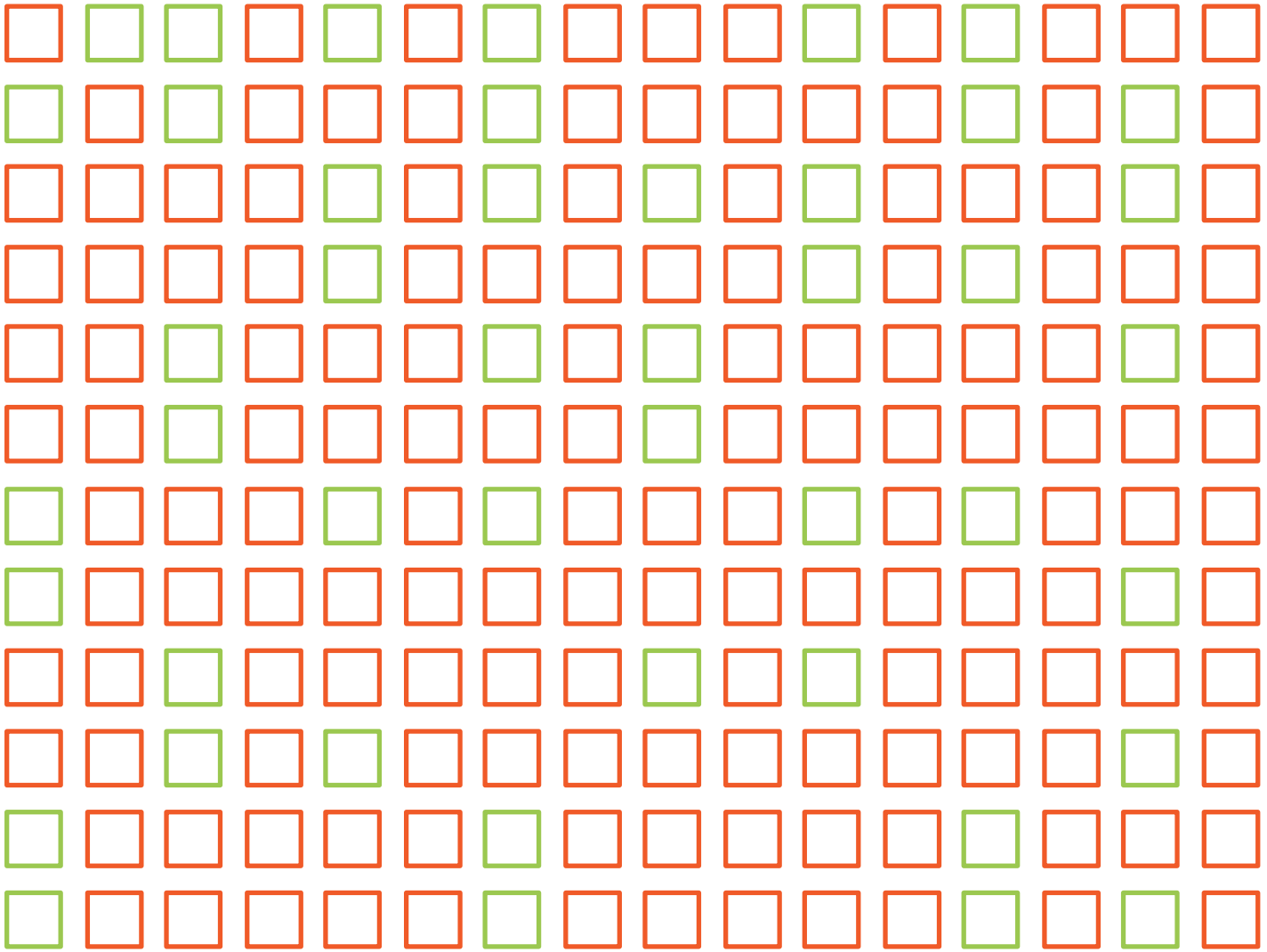
Finding Prime Numbers



Only 1 thead
is working

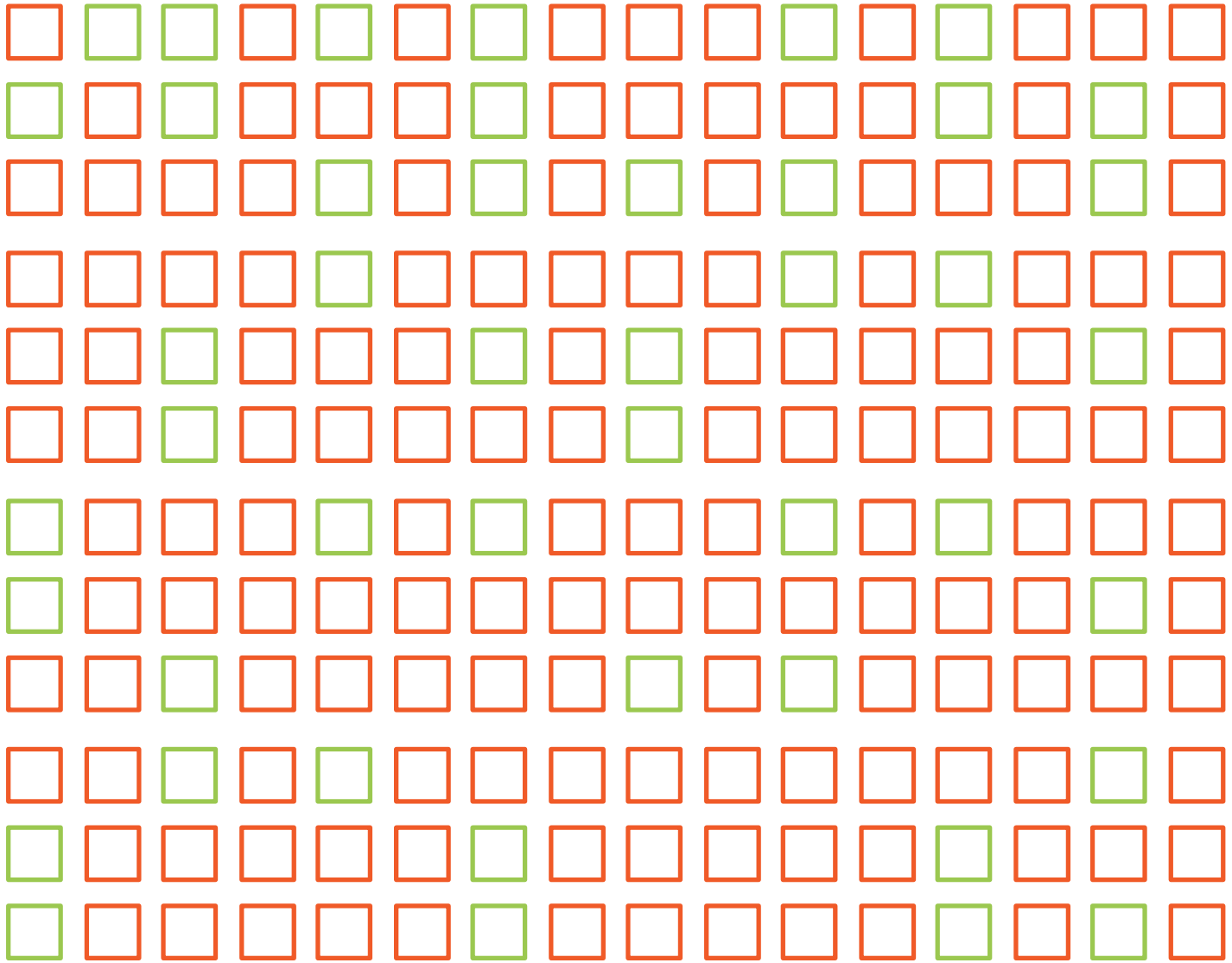
Meaning only
1 core
is used...

Finding Prime Numbers



With 4 cores
we could go faster!

Finding Prime Numbers



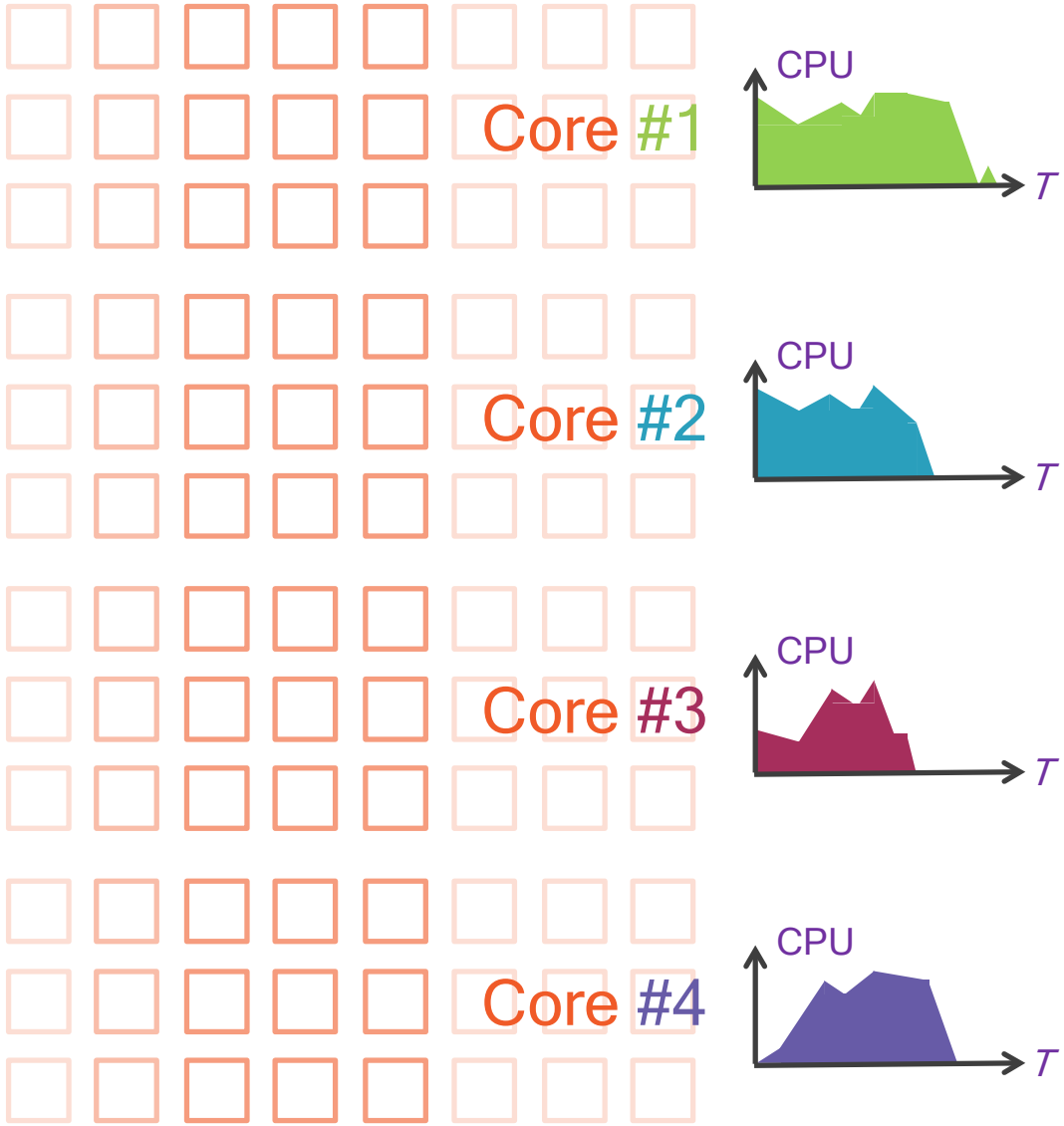
Core #1

Core #2

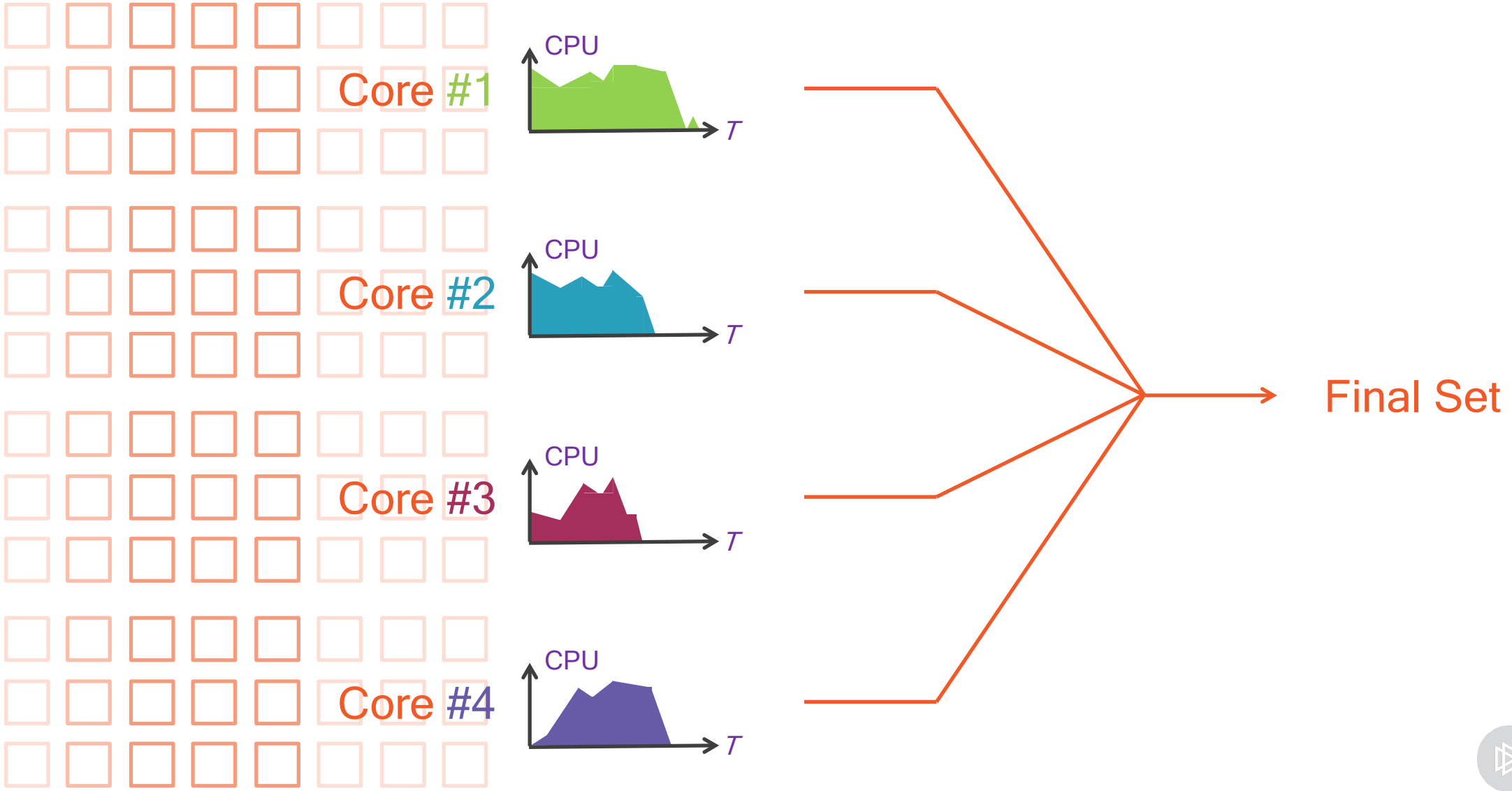
Core #3

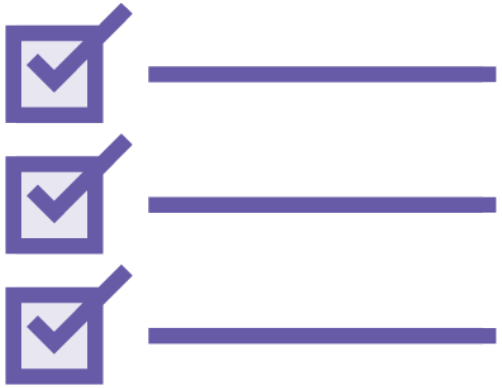
Core #4

Finding Prime Numbers



Finding Prime Numbers





We need a way to distribute the computation on several threads

We need to know when all the threads have finished their task

We need to launch a post-processing at that moment

```
Callable<List<Integer>> task = () -> findPrimes(inputSet);
```

First we need a task that takes a set of numbers

And returns the set of prime numbers among them

This task is a Callable

Real Life Example of Cyclic Barrier:

A Multithreaded download manager. The download manager will start multiple threads to download each part of the file simultaneously and also assume the files are downloaded from P2P. But here, suppose that you want the intergity check for the downloaded pieces to be done after a particular time interval. Here cyclic barrier plays an important role. After each time interval, each thread will wait at the barrier so that thread associated with cyclic barrier can do the integrity check. This integrity check can be done multiple times thanks to CyclicBarrier

```
CyclicBarrier barrier = new CyclicBarrier(4);
```

This callable has to wait for the other tasks launched in parallel, when its task is done

For that, we create a CyclicBarrier object

The parameter is the number of tasks that will be launched

```
Callable<List<Integer>> task = () -> {
   Set<Integer> result = findPrimes(inputSet);
   try {
      barrier.await(); // Blocks until everybody is ready
   } catch (Exception e) {...}
   return result;
}
```

```
Callable<List<Integer>> task = () -> {
   Set<Integer> result = findPrimes(inputSet);
   try {
      barrier.await(); // Blocks until everybody is ready
   } catch (Exception e) {...}
   return result;
}
```

```
Callable<List<Integer>> task = () -> {
   Set<Integer> result = findPrimes(inputSet);
   try {
      barrier.await(); // Blocks until everybody is ready
   } catch (Exception e) {...}
   return result;
}
```

```
Callable<List<Integer>> task = () -> {
   Set<Integer> result = findPrimes(inputSet);
   try {
      barrier.await(); // Blocks until everybody is ready
   } catch (Exception e) {...}
   return result;
}
```

```
Callable<List<Integer>> task = () -> {
   Set<Integer> result = findPrimes(inputSet);
   try {
      barrier.await(); // Blocks until everybody is ready
   } catch (Exception e) {...}
   return result;
}
```

We then tell the callable to wait for the barrier to open

```
Callable<List<Integer>> task = () -> {
   Set<Integer> result = findPrimes(inputSet);
   try {
      barrier.await(); // Blocks until everybody is ready
   } catch (Exception e) {...}
   return result;
}
```

We then tell the callable to wait for the barrier to open

The await call blocks...

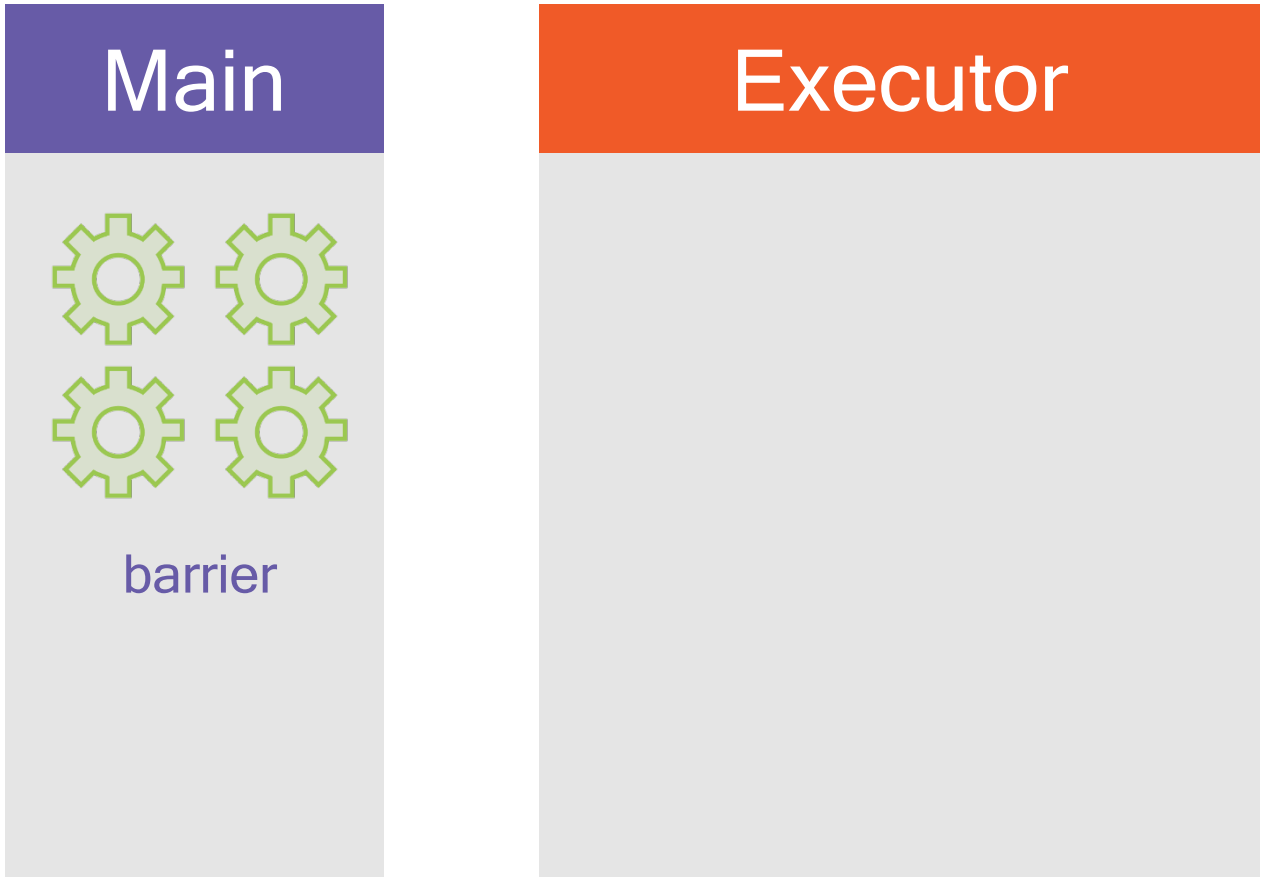
```
Callable<List<Integer>> task = () -> {
   Set<Integer> result = findPrimes(inputSet);
   try {
      barrier.await(); // Blocks until everybody is ready
   } catch (Exception e) {...}
   return result;
}
```

We then tell the callable to wait for the barrier to open

The await call blocks...

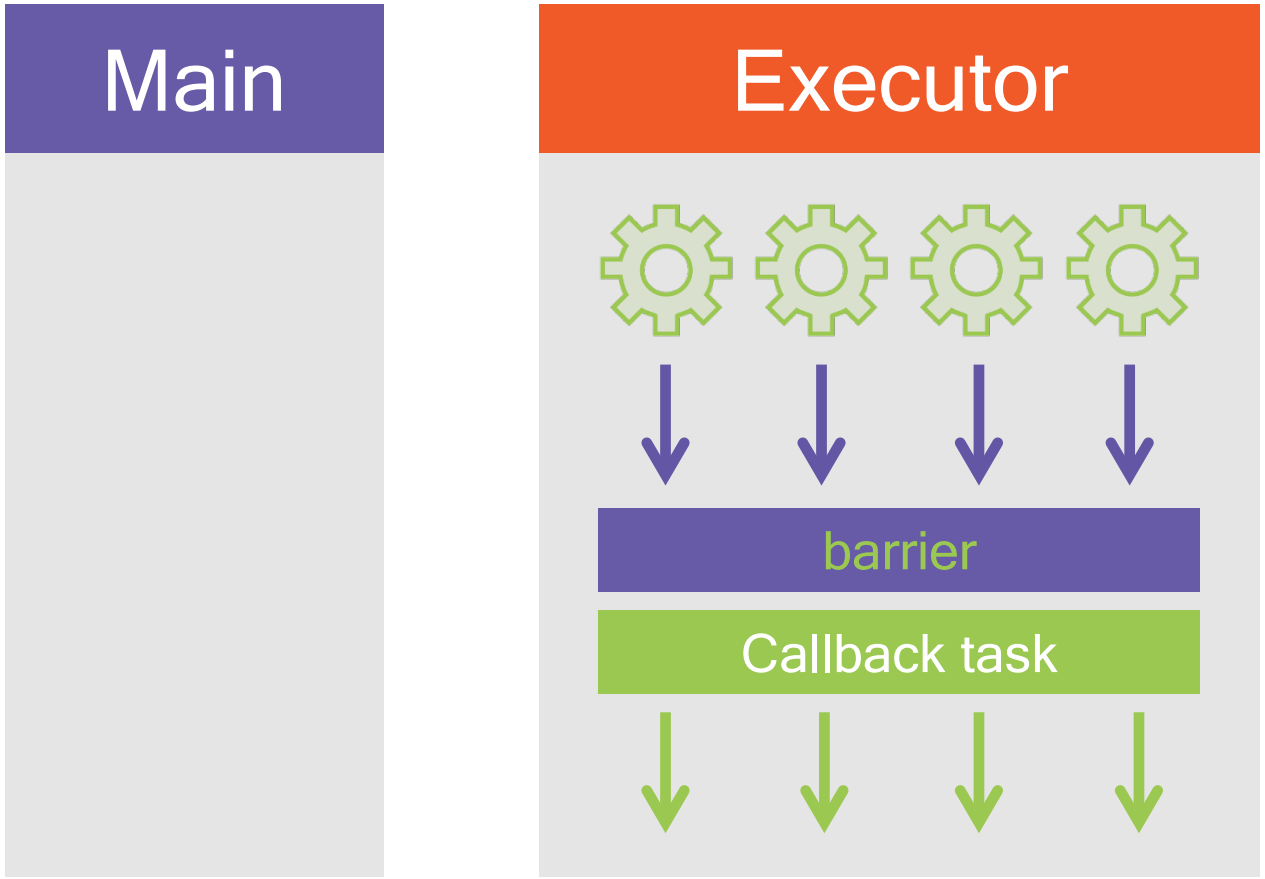
... until 4 calls have been made on it

How Does the CyclicBarrier Work?



executor.submit()

How Does the CyclicBarrier Work?



```
public class Worker implements Callable<List<Integer>> {

  private CyclicBarrier barrier;
  private List<Integer> inputList;

  public void Worker(CyclicBarrier barrier, List<Integer> inputList) {
    this.barrier = barrier;
    this.inputList = inputList;
  }

  public List<Integer> call() {
    List<Integer> result = findPrimes(inputList);
    try {
      barrier.await();
    } catch(InterruptedException | BrokenBarrierException e) {
      // Error handling
    }
    return result;
  }
}
```

```
public class Worker implements Callable<List<Integer>> {

  private CyclicBarrier barrier;
  private List<Integer> inputList;

  public void Worker(CyclicBarrier barrier, List<Integer> inputList) {
    this.barrier = barrier;
    this.inputList = inputList;
  }

  public List<Integer> call() {
    List<Integer> result = findPrimes(inputList);
    try {
      barrier.await();
    } catch(InterruptedException | BrokenBarrierException e) {
      // Error handling
    }
    return result;
  }
}
```

```
public class Worker implements Callable<List<Integer>> {

  private CyclicBarrier barrier;
  private List<Integer> inputList;

  public void Worker(CyclicBarrier barrier, List<Integer> inputList) {
    this.barrier = barrier;
    this.inputList = inputList;
  }

  public List<Integer> call() {
    List<Integer> result = findPrimes(inputList);
    try {
      barrier.await();
    } catch(InterruptedException | BrokenBarrierException e) {
      // Error handling
    }
    return result;
  }
}
```

```
CyclicBarrier barrier = new CyclicBarrier(4);
ExecutorService service = Executors.newFixedThreadPool(4);

Worker worker1 = new Worker(barrier, inputList1);
// More workers

Future<List<Integer>> future1 = service.submit(worker1);
// More submissions

List<Integer> finalResult = new ArrayList<>(future1.get());
finalResult.addAll(future2.get());
// More results
```

```
CyclicBarrier barrier = new CyclicBarrier(4);
ExecutorService service = Executors.newFixedThreadPool(4);

Worker worker1 = new Worker(barrier, inputList1);
// More workers

Future<List<Integer>> future1 = service.submit(worker1);
// More submissions

List<Integer> finalResult = new ArrayList<>(future1.get());
finalResult.addAll(future2.get());
// More results
```

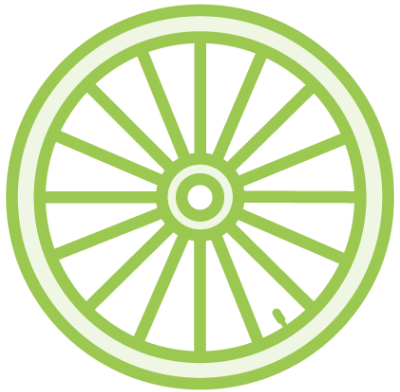
```
CyclicBarrier barrier = new CyclicBarrier(4);
ExecutorService service = Executors.newFixedThreadPool(4);

Worker worker1 = new Worker(barrier, inputList1);
// More workers

Future<List<Integer>> future1 = service.submit(worker1);
// More submissions

List<Integer> finalResult = new ArrayList<>(future1.get());
finalResult.addAll(future2.get());
// More results
```

```
CyclicBarrier barrier = new CyclicBarrier(4);
ExecutorService service = Executors.newFixedThreadPool(4);

Worker worker1 = new Worker(barrier, inputList1);
// More workers

Future<List<Integer>> future1 = service.submit(worker1);
// More submissions

List<Integer> finalResult = new ArrayList<>(future1.get());
finalResult.addAll(future2.get());
// More results
```

A CyclicBarrier is cyclic because it can be reused without resetting. So in normal usage, once all the threads are collected and the barrier is broken,it resets itself and can be used again.

From the Javadoc for reset() --> Resets the barrier to its initial state. If any parties are currently waiting at the barrier, they will return with a BrokenBarrierException.

So reset causes any currently waiting threads to throw a BrokenBarrierException and wake immediately. reset is used when you want to "break" the barrier.



Note also the caveat - once the threads have been awoken forcibly it's tricky to synchronize them again.

Note: you should never need to use reset() in normal circumstances.

The await() call is blocking

There are two versions:

- await()
- await(time, timeUnit)

Once opened a barrier is normally reset

The reset() method resets the barrier exceptionnally, causing the waiting tasks to throw a BrokenBarrierException

Handling of Errors

A BrokenBarrierException is raised if:

- a thread is interrupted while waiting
- the barrier is reset while some threads are waiting

CyclicBarrier

A tool to synchronize several threads between them, and let them continue when they reach a common point

A CyclicBarrier closes again once opened, allowing for cyclic computations, can also be reset

The threads can wait for each other on time outs

Latches

Real Life Use Case of Count Down Latch:

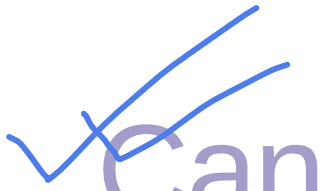
A Multithreaded download manager. The download manager will start multiple threads to download each part of the file simultaneously.(Provided the server supports multiple threads to download). Here each thread will call a countdown method of an instantiated latch. After all the threads have finished execution, the thread associated with the countdown latch will integrate the parts found in the different pieces together into one file

Stating the Problem

We need to start our application

An AuthenticationService, a DataService and an OrderService

Before serving clients, our application needs to make sure that several resources are properly initialized

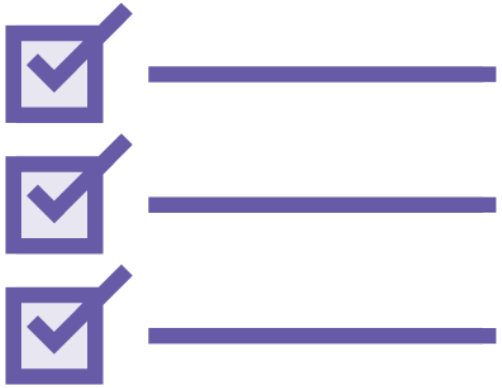


Can We Use a CyclicBarrier?

It seems that we can use a CyclicBarrier

But once all the services are available and our application starts…

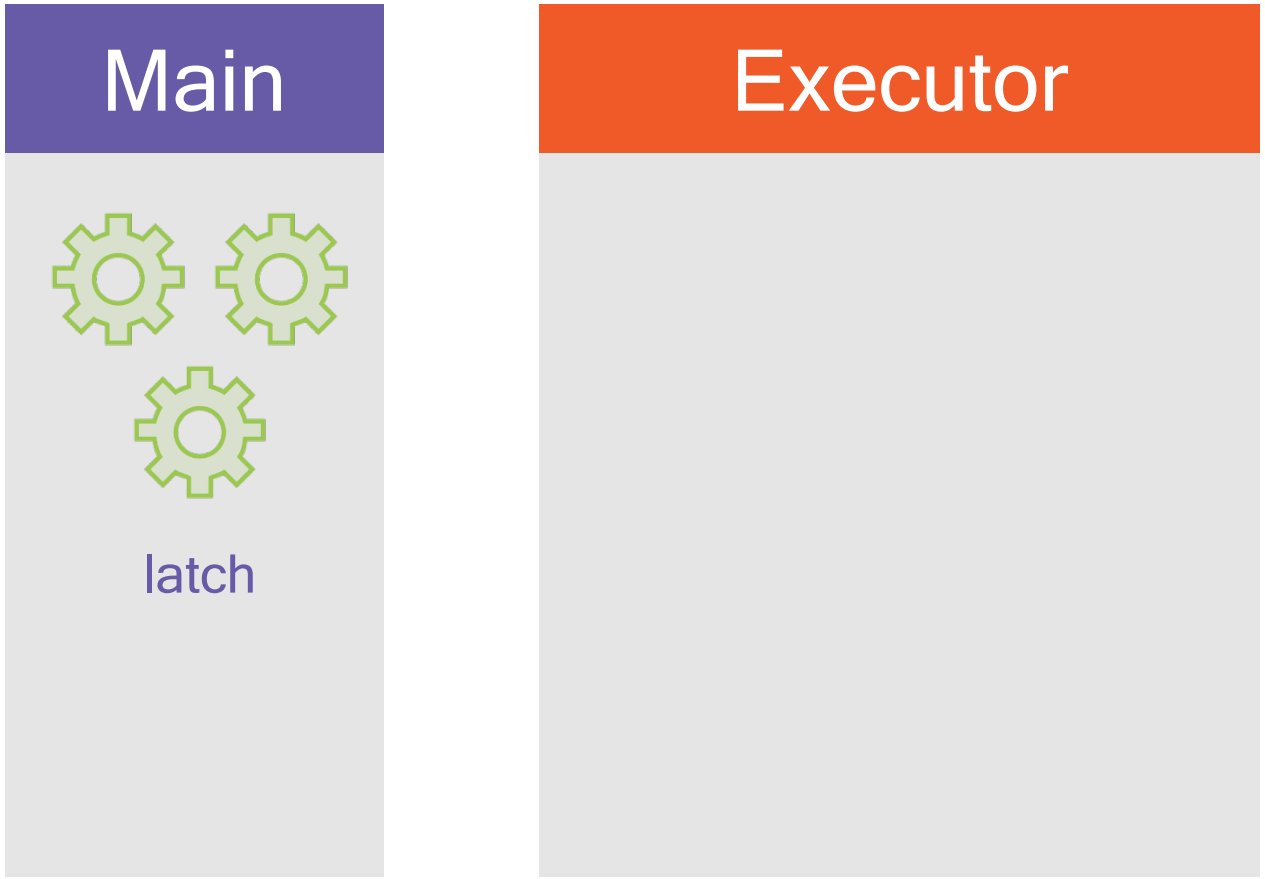
…we do not want the barrier to reset, thus blocking everything!



We need a kind of barrier that, once opened, cannot be closed

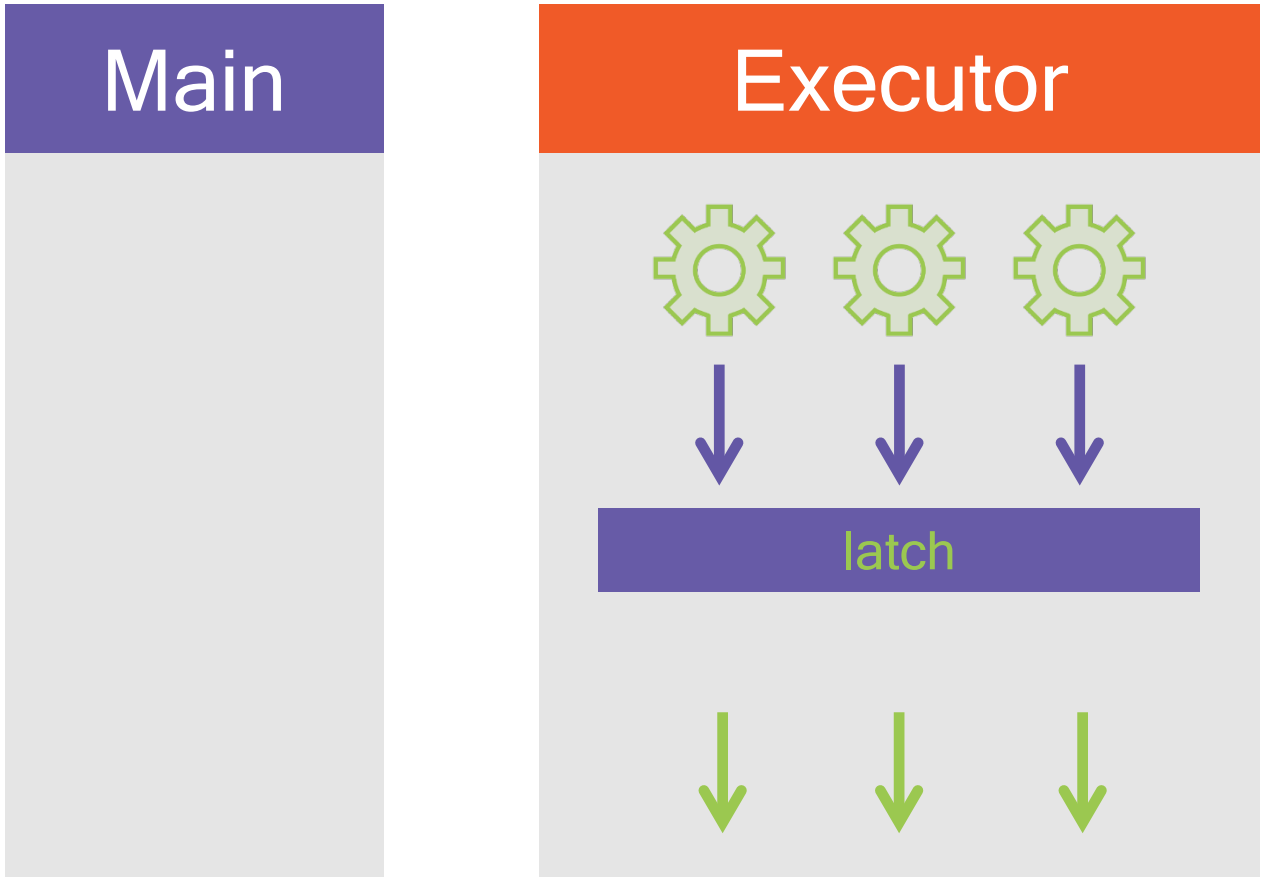
This is the countdown latch

How Does the CountDownLatch Work?



executor.submit()

How Does the CountDownLatch Work?



```
public class ServiceWorker implements Callable<List<Integer>> {

   private CountDownLatch latch;
   private Service service;

   public boolean Worker(CountDownLatch latch, Service service) {
      this.latch = latch;
      this.service = service;
   }

   public void call() {
      service.init();

      latch.countDown();
   }
}
```

```
public class ServiceWorker implements Callable<List<Integer>> {

   private CountDownLatch latch;
   private Service service;

   public boolean Worker(CountDownLatch latch, Service service) {
      this.latch = latch;
      this.service = service;
   }

   public void call() {
      service.init();

      latch.countDown();
   }
}
```

```
public class ServiceWorker implements Callable<List<Integer>> {

   private CountDownLatch latch;
   private Service service;

   public boolean Worker(CountDownLatch latch, Service service) {
      this.latch = latch;
      this.service = service;
   }

   public void call() {
      service.init();

      latch.countDown();
   }
}
```

```java
public class ServiceWorker implements Callable<List<Integer>> {

    private CountDownLatch latch;
    private Service service;

    public boolean Worker(CountDownLatch latch, Service service) {
        this.latch = latch;
        this.service = service;
    }

    public void call() {
        service.init();

        latch.countDown();
    }
}
```

```
CountDownLatch latch = new CountDownLatch(3);
ExecutorService executor = Executors.newFixedThreadPool(4);

ServiceWorker worker1 = new Worker(latch, dataService);
// More workers

Future<Boolean> future1 = executor.submit(worker1);
// More submissions

try {
   latch.await(10, TimeUnit.SECONDS); // blocks until the count reaches 0
   server.start();
} catch(InterruptedException e) {
   // Error handling
}
```

```
CountDownLatch latch = new CountDownLatch(3);
ExecutorService executor = Executors.newFixedThreadPool(4);

ServiceWorker worker1 = new Worker(latch, dataService);
// More workers

Future<Boolean> future1 = executor.submit(worker1);
// More submissions

try {
   latch.await(10, TimeUnit.SECONDS); // blocks until the count reaches 0
   server.start();
} catch(InterruptedException e) {
   // Error handling
}
```

```
CountDownLatch latch = new CountDownLatch(3);
ExecutorService executor = Executors.newFixedThreadPool(4);

ServiceWorker worker1 = new Worker(latch, dataService);
// More workers

Future<Boolean> future1 = executor.submit(worker1);
// More submissions

try {
   latch.await(10, TimeUnit.SECONDS); // blocks until the count reaches 0
   server.start();
} catch(InterruptedException e) {
   // Error handling
}
```

```
CountDownLatch latch = new CountDownLatch(3);
ExecutorService executor = Executors.newFixedThreadPool(4);

ServiceWorker worker1 = new Worker(latch, dataService);
// More workers

Future<Boolean> future1 = executor.submit(worker1);
// More submissions

try {
   latch.await(10, TimeUnit.SECONDS); // blocks until the count reaches 0
   server.start();
} catch(InterruptedException e) {
   // Error handling
}
```

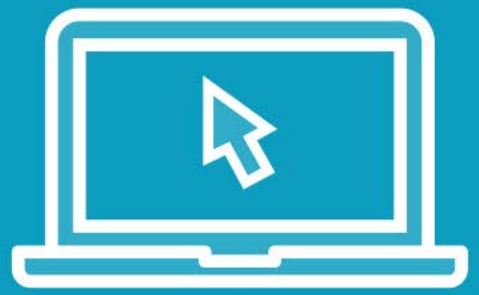
CountDownLatch

A tool to check that different threads did their task

And synchronize the beginning of subsequent tasks on the last one to complete

Once open CountDownLatch cannot be closed again

Demo



Let us see some code!

Let us see a barrier in action

Demo Wrapup

What did we see?

How to create barriers with callbacks to have threads wait for each other

How to set a time out on the CyclicBarrier.await() call

How to set a time out and a cancel on a Future.get() call

Module Wrapup

What did we learn?

There are two tools to trigger an action on the completion of other actions

The CyclicBarrier: useful for parallel computations

The CountDownLatch: useful for starting an application on the completion of different initializations