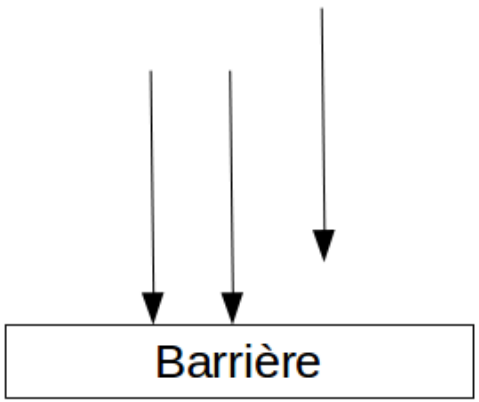
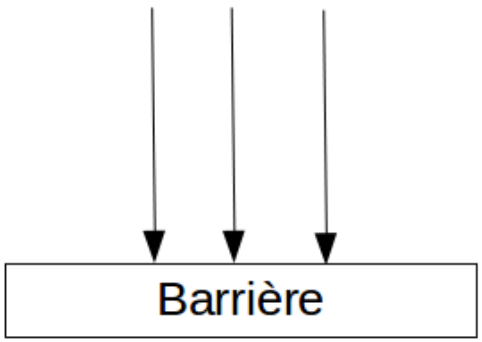
# Programmation Répartie : Mécanismes de synchronisation

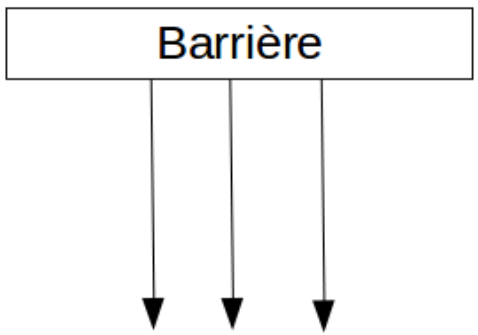
## Coordination entre threads

- Comment des threads peuvent-ils se **coordonner** ? Comment les faire attendre à un même point de rendez-vous ?
- Avec `join`, il est possible d'attendre un thread, mais c'est la **fin** d'un thread qui est attendu
- Comment faire pour **attendre** une **partie du calcul** d'un thread ?

# Coordination de threads



## Solutions

- Il est possible d'utiliser le mécanisme de *wait/notify*, couplé avec un compteur
- En Java, il existe des solutions dans la bibliothèque standard pour faciliter ce type de synchronisation
  - `CountDownLatch`
  - `CyclicBarrier`
  - `Phaser`

# CountDownLatch

## Utilisation

- Permet à des threads d'attendre qu'un ensemble de threads termine un ensemble de tâches
- Ce loquet est initialisé avec une valeur, qui sert de **compteur** de thread.
- A chaque fois qu'un thread termine sa tâche, il le signale et le compteur est décrémenté. Lorsque le compteur arrive à zéro, les threads en attente poursuivent leur exécution.
- Les threads peuvent donc soit
  - **attendre**
  - **signaler** leur arrivée

Une fois le compteur arrivé à 0, le loquet ne peut pas être réutilisé

## En Java

- `CountDownLatch loquet = new CountDownLatch(N)`
- `await()` : le thread est mis en attente jusqu'à ce que le compteur du loquet atteigne 0
- `countDown()` : décrémente le compteur. Si le compteur atteint 0, tous les threads en attente sont réveillés

`https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountDownLatch.html`

## Driver

```
import java.util.concurrent.CountDownLatch;
class Driver {
      static int N = 5;
      public static void main(String args[]) throws InterruptedException {
      CountDownLatch startSignal = new CountDownLatch(1);
      CountDownLatch doneSignal = new CountDownLatch(N);

      for (int i = 0; i < N; ++i) // cree et lance les N threads
        new Thread(new Worker(startSignal, doneSignal)).start();

      //doSomething(); // code execute avant le debut du code des threads
      try{
          Thread.sleep(5000);
      }
      catch(Exception e){e.printStackTrace();}

      startSignal.countDown(); // les threads peuvent executer leur code
      //doSomethingElse();
      doneSignal.await(); // attente des N threads
    }
 }
```

## Worker

```
import java.util.concurrent.CountDownLatch;
class Worker implements Runnable {
    private final CountDownLatch startSignal;
    private final CountDownLatch doneSignal;
    Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
        this.startSignal = startSignal;
        this.doneSignal = doneSignal;
    }
    public void run() {
        try {
            startSignal.await(); //attente du signal de depart
            doWork();
            doneSignal.countDown(); //signale que l'on a fini
        } catch (InterruptedException ex) {}
    }

    void doWork() { System.out.println("Thread travaille"); }
}
```

# CyclicBarrier

La barrière est proche du concept du loquet, elle permet également la coordination entre threads

## CyclicBarrier

- La barrière contient également un compteur de thread
- La barrière est initialisée à un **nombre de thread** à atteindre avant qu'elle ne s'ouvre
- Un thread signale son arrivée **et** attend l'ouverture de la barrière. Il n'y a pas deux comportements différents comme avec le loquet

## Threads et CyclicBarrier

- Les threads **arrivent** à cette barrière, et se mettent en **attente** tant que le nombre n'est pas atteint
- Lorsque le nombre de thread est atteint, la barrière s'**ouvre**, et les threads en attente sur celle-ci poursuivent leur exécution
- La barrière est ensuite **reinitialisée**, et peut être réutilisée.
- Il est possible d'ajouter une **action** à réaliser lors de l'**ouverture** de la barrière, à l'aide d'un thread à déclarer au constructeur

## Java

- `CyclicBarrier barriere = new CyclicBarrier(N)` : création d'une nouvelle barrière, qui ne s'ouvre que lorsque N threads sont arrivés
- `await()` : Le thread se met en attente sur la barrière. La barrière enregistre cette arrivée

`https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html`

## Main

```
import java.util.concurrent.CyclicBarrier;

public class Main{

    public static void main(String args[]){

        final CyclicBarrier cb = new CyclicBarrier(3);

        Thread t1 = new Thread(new Worker(cb));
        Thread t2 = new Thread(new Worker(cb));
        Thread t3 = new Thread(new Worker(cb));

        t1.start();
        t2.start();
        t3.start();
    }

}
```

## Worker

```
import java.util.concurrent.CyclicBarrier;
public class Worker implements Runnable {

    private CyclicBarrier barrier;

    public Worker(CyclicBarrier barrier) {
        this.barrier = barrier;
    }

    public void run() {
        try {
            while(true){
                System.out.println(Thread.currentThread().getName() + " est en attente a la barriere");
                barrier.await();
                System.out.println(Thread.currentThread().getName() + " a franchi la barriere");
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

La classe *Phaser* est également un outil de coordination entre threads, mais plus flexible

## Phaser

- Contrairement au *CountdownLatch* et à la *CyclicBarrier*, le nombre de thread n'a pas besoin d'être connu à l'avance. Un thread **s'enregistre** auprès de ce *phaser* pour indiquer qu'il participe à la coordination, et peut ensuite se désinscrire
- Un *phaser* est **réutilisable** comme la *CyclicBarrier*
- Un thread lorsqu'il arrive à une étape de synchronisation se signale, et peut soit **attendre** que tous les threads soient arrivés, soit **poursuivre** son exécution

## Java

- `register()` : le thread s'enregistre auprès du Phaser
- `arrive()` : le thread signale son arrivée, et poursuit son éxécution
- `arriveAndAwaitAdvance()` : le thread signale son arrivée et attend les autres
- `arriveAndDeregister()` : le thread signale son arrivée, se désinscrit, et poursuit son éxécution

`https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Phaser.html`

## Main

```
import java.util.concurrent.Phaser;

public class PhaserExample
{
    public static void main(String[] args) throws InterruptedException
    {
                Phaser phaser = new Phaser();
                phaser.register();
                (new Thread(new Worker(phaser))).start();
                (new Thread(new Worker(phaser))).start();
                (new Thread(new Worker(phaser))).start();

                phaser.arrive();
                System.out.println(Thread.currentThread().getName()+" arrive");

    }
}
```

## Worker

```
import java.util.concurrent.Phaser;


public class Worker implements Runnable{


    Phaser phaser;

    public Worker(Phaser p){
        phaser = p;
        phaser.register();
    }

    public void run(){
        //doSomeWork();
         phaser.arriveAndAwaitAdvance();
         System.out.println(Thread.currentThread().getName()+" arrive");
         //doOtherWork();
    }

}
```

## Gestion des phases

- Le phaser maintient un numéro de phase (d'étape)
- Chaque génération de phaser à un numéro. Au départ il est à 0, et s'incrémente quand tous les threads sont arrivés
- Il est possible de préciser quel numéro de phase le thread doit attendre

# Conclusion

## CountDownLatch

- Outil de synchronisation non réutilisable
- Le comportement d'attente et de signalement d'arrivée est séparé dans deux méthodes

## CyclicBarrier

- Outil de synchronisation réutilisable
- L'attente et le signalement d'arrivée sont effectués par la même méthode

## Phaser

- Outil de synchronisation réutilisable
- Les threads ont la possibilité de poursuivre leur exécution ou d'attendre au point de synchronisation
- Nombre de thread variable
- Gestion des phases