<!--DAM-->

Mecanismos adicionales de sincronización en Java {

<Por="Miguel, Ruben e Isma"/>

}

Contenidos

Concepto {

CountDownLatch es un mecanismo de sincronización en Java que permite a un hilo esperar hasta que un conjunto de operaciones realizadas por otros hilos se complete. Funciona mediante un contador que se inicializa con un valor determinado y se decrementa a medida que los hilos completan sus operaciones. El hilo que espera en el CountDownLatch bloquea su ejecución hasta que el contador llega a cero.

```
Task 3 en progreso...
Task 2 en progreso...
Task 1 en progreso...
Task 2 completada.
Task 3 completada.
Task 1 completada.
Todas las tareas han finalizado. Continuar con la siguiente fase.

Process finished with exit code 0
```

```java
public class CountDownLatchh {
        public static void main(String[] args) throws InterruptedException {
            CountDownLatch latch = new CountDownLatch(3);

            // Hilos simulando tareas
            Thread task1 = new Thread(new Task(latch,  taskName: "Task 1"));
            Thread task2 = new Thread(new Task(latch,  taskName: "Task 2"));
            Thread task3 = new Thread(new Task(latch,  taskName: "Task 3"));

            task1.start();
            task2.start();
            task3.start();

            // Esperar a que todas las tareas terminen
            latch.await();

            System.out.println("Todas las tareas han finalizado. Continuar con la siguiente fase.");
        }
    }

    3 usages
    class Task implements Runnable {
        2 usages
        private final CountDownLatch latch;
        3 usages
        private final String taskName;

        3 usages
        Task(CountDownLatch latch, String taskName) {
            this.latch = latch;
            this.taskName = taskName;
        }

        @Override
        public void run() {
            // Simular la ejecución de la tarea
            System.out.println(taskName + " en progreso...");
            try {
                Thread.sleep( l: 2000); // Simulando el tiempo de ejecución
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(taskName + " completada.");
            latch.countDown(); // Decrementar el contador
        }
    }
```

}

Incorporación y casos de uso {

Incorporación en el Framework:

CountDownLatch está incorporado en el framework estándar de Java (java.util.concurrent package), por lo que no es necesario utilizar una librería externa.

Casos de Uso:

Esperar a que varios hilos completen una tarea antes de continuar con otra parte del programa.
Coordinar la ejecución de hilos en un escenario donde se necesita que todos finalicen antes de proceder.

}

Pros y contras {

Ventajas:

Facilita la sincronización y coordinación entre hilos.
Incorporado en el framework estándar, evitando dependencias externas innecesarias.

Inconvenientes:

Limitado a situaciones donde la espera se basa en la finalización de un número específico de operaciones.

}

Concepto {

CyclicBarrier es otro mecanismo de sincronización en Java que permite a un conjunto de hilos esperar unos a otros en un punto de encuentro específico antes de continuar con su ejecución. A diferencia de CountDownLatch, un CyclicBarrier se puede reutilizar después de que se haya alcanzado el punto de encuentro.

```
Task 1 en progreso...
Task 2 en progreso...
Task 3 en progreso...
Task 2 completada.
Task 3 completada.
Task 1 completada.
Todas las tareas han finalizado. Continuar con la siguiente fase.

Process finished with exit code 0
```

```java
public class CyclicBarrierr {
    public static void main(String[] args) {
        // Punto de encuentro para tres hilos
        CyclicBarrier barrier = new CyclicBarrier( parties: 3, () ->
                System.out.println("Todos los hilos han llegado al punto de encuentro. Continuar..."));

        // Hilos simulando tareas
        Thread task1 = new Thread(new Taskk(barrier,  taskName: "Task 1"));
        Thread task2 = new Thread(new Taskk(barrier,  taskName: "Task 2"));
        Thread task3 = new Thread(new Taskk(barrier,  taskName: "Task 3"));

        task1.start();
        task2.start();
        task3.start();
    }
}

3 usages
class Taskk implements Runnable {
    2 usages
    private final CyclicBarrier barrier;
    4 usages
    private final String taskName;

    3 usages
    Taskk(CyclicBarrier barrier, String taskName) {
        this.barrier = barrier;
        this.taskName = taskName;
    }

    @Override
    public void run() {
        // Simular la ejecución de la tarea
        System.out.println(taskName + " en progreso...");
        try {
            Thread.sleep( t: 2000); // Simulando el tiempo de ejecución
            System.out.println(taskName + " ha llegado al punto de encuentro.");
            barrier.await(); // Esperar a que los demás hilos lleguen al punto de encuentro
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
        System.out.println(taskName + " continuando después del punto de encuentro.");
    }
}
```

}

Incorporación y casos de uso {

Incorporación en el Framework:

CyclicBarrier también está incorporado en el framework estándar de Java (java.util.concurrent package), por lo que no es necesario utilizar una librería externa.

Casos de Uso:

Dividir una tarea en sub-tareas y ejecutarlas en paralelo, esperando a que todas terminen antes de continuar.
Coordinar múltiples hilos en un punto de sincronización común.

}

Pros y contras {

Ventajas:

Permite la sincronización cíclica entre hilos.
Reutilizable para situaciones donde se necesite repetir la sincronización.

Inconvenientes:

Limitado a situaciones donde se desea que los hilos se sincronicen en un punto específico.

}

<!--Dam-->

Gracias {

<Por="Miguel, Ruben e Isma"/>

}