# Locks de Lectura y Escritura

### *Interfaz ReadWriteLock y su implementación*

## Integrantes del Grupo:

- Cabanne
- Gattas
- Kissner
- Occhi

## ¿Como funciona?

Un Lock de escritura y lectura mantiene un par de Locks asociados, uno para solo operaciones de lectura y otro para escritura. El Lock de lectura puede ser mantenido simultáneamente por múltiples hilos lectores **siempre y cuando no haya hilos escritores interfiriendo.**

Todas las implementaciones de **ReadWriteLock** deben garantizar que los efectos de sincronizar la memoria de las operaciones con el Lock de escritura, también se mantengan con respecto al Lock de lectura. Esto significa que, si un hilo adquiere exitosamente el Lock de escritura, el mismo hilo va a ver todas las actualizaciones hechas previas a la liberación del Lock de escritura.

## Ventajas vs el Lock de exclusión mutua (Mutex)

Un Lock de escritura y lectura permite un mayor nivel de concurrencia a la hora de acceder a datos compartidos que la que se permite normalmente por un Lock de exclusión mutua (Mutex). Se beneficia del hecho de que, mientras que solo un hilo a la vez puede modificar los datos compartidos, en muchos casos cualquier numero de hilos puede acceder concurrentemente a la leer la data. En teoría el incremento en la concurrencia permitido por el uso de un Lock de escritura y lectura va a decantar en una mejora del rendimiento sobre el uso de un Lock Mutex.

## ¿En que situaciones es recomendable su uso?

Si un Lock de escritura y lectura va a mejorar el rendimiento sobre un Mutex o no, va a depender de la frecuencia en la que los datos son leídos en comparación de la frecuencia en la que son modificados, la duración de las operaciones de escritura y lectura y la contención de los datos (el número de hilos que trataran de leer o escribir sobre los datos al mismo tiempo).

Por ejemplo: Si tenemos una colección de datos ya inicialmente cargada con datos y luego es muy poco frecuentemente modificada, mientras que es muy frecuentemente leída, es un caso candidato ideal para el uso de un Lock de escritura y lectura.

Sin embargo, si las actualizaciones sobre esta colección se vuelven mas frecuentes, entonces los datos van a pasar la mayor parte del tiempo siendo bloqueados

exclusivamente por el Lock de escritura, y habrá muy poca mejora en torno a la concurrencia con el uso del Mutex.

## Implementación en Java:

```java
public class RecursoCompartido {
    private final ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
    private final Lock writeLock = readWriteLock.writeLock();
    private final Lock readLock = readWriteLock.readLock();
    private int numAleatorio = 0;

    public void cambiarElemento(){
        writeLock.lock();
        try{
            this.numAleatorio = (int) (Math.random()*1000 + 1);
            System.out.println(x: "Se ha cambiado el numero");
        } finally {
            writeLock.unlock();
        }
    }

    public void leerElemento() throws InterruptedException{
        readLock.lock();
        try{
          System.out.println("El hilo "+Thread.currentThread().getName()+" lee "
                  + "el numero "+this.numAleatorio);
          Thread.sleep(millis: 2000);
        } finally{
          readLock.unlock();
        }
    }
}
```

```java
public class HiloLector implements Runnable{
    private RecursoCompartido unR = new RecursoCompartido();

    public HiloLector(RecursoCompartido aR ){
        this.unR = aR;
    }

    public void run(){
        while(true){
            try {
                this.unR.leerElemento();
                Thread.sleep(millis: 250);
            } catch (InterruptedException ex) {
                Logger.getLogger(name: HiloLector.class.getName()).log(level: Level.SEVERE, msg: null, thrown: ex);
            }
        }
    }
}
```

```java
public class HiloEscritor implements Runnable{
    private RecursoCompartido unR = new RecursoCompartido();

    public HiloEscritor(RecursoCompartido aR ){
        this.unR = aR;
    }

    public void run(){
        while(true){
            this.unR.cambiarElemento();
            try {
                Thread.sleep(millis: 750);
            } catch (InterruptedException ex) {
                Logger.getLogger(name:HiloLector.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
            }
        }
    }
}
```

```java
public class Main {
    public static void main(String args[]){
        RecursoCompartido rC = new RecursoCompartido();
        HiloEscritor pHE = new HiloEscritor(aR: rC);
        HiloLector pHL = new HiloLector(aR: rC);

        Thread t1 = new Thread(target: pHE,name: "Escritor");
        Thread t2 = new Thread(target: pHL,name: "Lector-1");
        Thread t3 = new Thread(target: pHL,name: "Lector-2");
        Thread t4 = new Thread(target: pHL,name: "Lector-3");
        Thread t5 = new Thread(target: pHL,name: "Lector-4");
        Thread t6 = new Thread(target: pHL,name: "Lector-5");

        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();
        t6.start();
    }
}
```