# JUEGO DE LA SERPIENTE

**AUTORES:**

YORKS GOMEZ – CESAR VÁSQUEZ

**PROFESOR:**

JAVIER IVAN TOQUICA

ESCUELA COLOMBIANA DE INGENIERÍA JULIO GARAVITO

ARQUITECTURAS DE SOTFWARE – GRUPO # 1

INGENIERÍA DE SISTEMAS

BOGOTÁ D.C.

2023

# INTRODUCCIÓN

En este informe de laboratorio continuaremos trabajando con los conceptos principales como lo son la concurrencia vista o implementada por medio de hilos. En adición usaremos funciones o estados de los hilos de forma correcta, dentro de algunas funcionalidades podemos encontrar mecanismos de sincronización, usaremos esta técnica para asegurar que los hilos que se ejecuten al tiempo en un orden predeterminado y que se use de forma correcta los recursos compartidos por más de un hilo, es decir, conocer la condición de carrera y determinar una región crítica correcta.

## DESARROLLO

### Parte l

Se necesita modificar la aplicación de manera que cada t milisegundos de ejecución de los threads, se detengan todos los hilos y se muestre el número de primos encontrados hasta el momento. Luego, se debe esperar a que el usuario presione ENTER para reanudar la ejecución de estos. Utilice los mecanismos de sincronización provistos por el lenguaje (wait y notify, notifyAll).

```
@Override
public void run() {
    for(int i = 0;i < NTHREADS;i++ ) {
        pft[i].start();
    }

    try {
        while(true) {
            this.sleep(TMILISECONDS);

            PrimeFinderThread.setMustWait(mustWait: true);

            for (PrimeFinderThread primeFinderThread : pft)

                if(!primeFinderThread.isInterrupted())
                    this.sleep(millis: 20);

            System.out.println("Amount of primes found is: " + PrimeFinderThread.getCounter());
            new Scanner(System.in).nextLine();

            PrimeFinderThread.setMustWait(mustWait: false);
            synchronized(this) {
                this.notifyAll();
            }

        }
    } catch(Exception ex) {
        ex.printStackTrace();
        System.exit(status: 0);
    }
}
```

```java
    @Override
public void run(){
        for (int i= a;i < b;i++){
            if (isPrime(i)){
                primes.add(i);

                synchronized(counter) {
                    counter++;
                }

                System.out.println(i);

                try {
                    if(mustWait) {
                        synchronized (control) {
                            control.wait();
                        }
                    }
                } catch(Exception ex) {
                    ex.printStackTrace();
                    System.exit(status: 0);
                }
            }
        }
}
```

Usamos nuevos atributos en la clase PrimeFinderThread, para contar los primos inicialmente en 0, un booleano para saber si tiene que esperar el hilo inicialmente no y obviamente nuestro hilo Control con el que habrá condición de carrera.

```java
private static Integer counter = 0;
private static boolean mustWait = false;
private Control control;
```

Modificamos el método run() de la clase PrimeFinderThread, empezamos sincronizamos el contador.

```java
synchronized(counter) {
    counter++;
}
```

Modificamos el método run() de la clase Control, en donde iniciamos haciendo que el hilo que se está ejecutando es decir Control, actualmente entre en reposo durante el número especificado de milisegundos en este caso nuestro atributo estático con valor de 5000.

Después ponemos en verdadero la bandera mustWait de si debe esperar el hilo,

```
try {
    while(true) {
        this.sleep(TMILISECONDS);

        PrimeFinderThread.setMustWait(mustWait: true);
```

Cuando cambie la bandera mustWait sincronizaremos el hilo Control para que se ponga en espera, pues quiere decir que ya pasaron los TMILISECONDS por lo cual debe esperar hasta que el usuario presione ENTER para continuar.

```
try {
    if(mustWait) {
        synchronized (control) {
            control.wait();
        }
    }
} catch(Exception ex) {
    ex.printStackTrace();
    System.exit(status: 0);
}
```

Luego simplemente cuando se acaba el tiempo de ejecución de los threads y se detienen los mismo, imprimimos el mensaje de cuantos primos se tienen hasta dicho momento y esperamos hasta que el usuario oprima Enter y así poder continuar.

```
for (PrimeFinderThread primeFinderThread : pft)

    if(!primeFinderThread.isInterrupted())
        this.sleep(millis: 20);

System.out.println("Amount of primes found is: " + PrimeFinderThread.getCounter());
new Scanner(System.in).nextLine();
```

Luego de que el usuario oprima el botón, cambiamos el valor de nuestro atributo mustWait a false, porque ya no se debería esperar y luego notificamos a todos los hilos para que funcionen nuevamente.

```
PrimeFinderThread.setMustWait(mustWait: false);
synchronized(this) {
    this.notifyAll();
}
```

**Parte II**

1. Cada serpiente extiende de Thread, por lo que cada serpiente es un hilo, esto es lo que las hace independientes entre sí.
2. 

   **Posibles condiciones de carrera:**

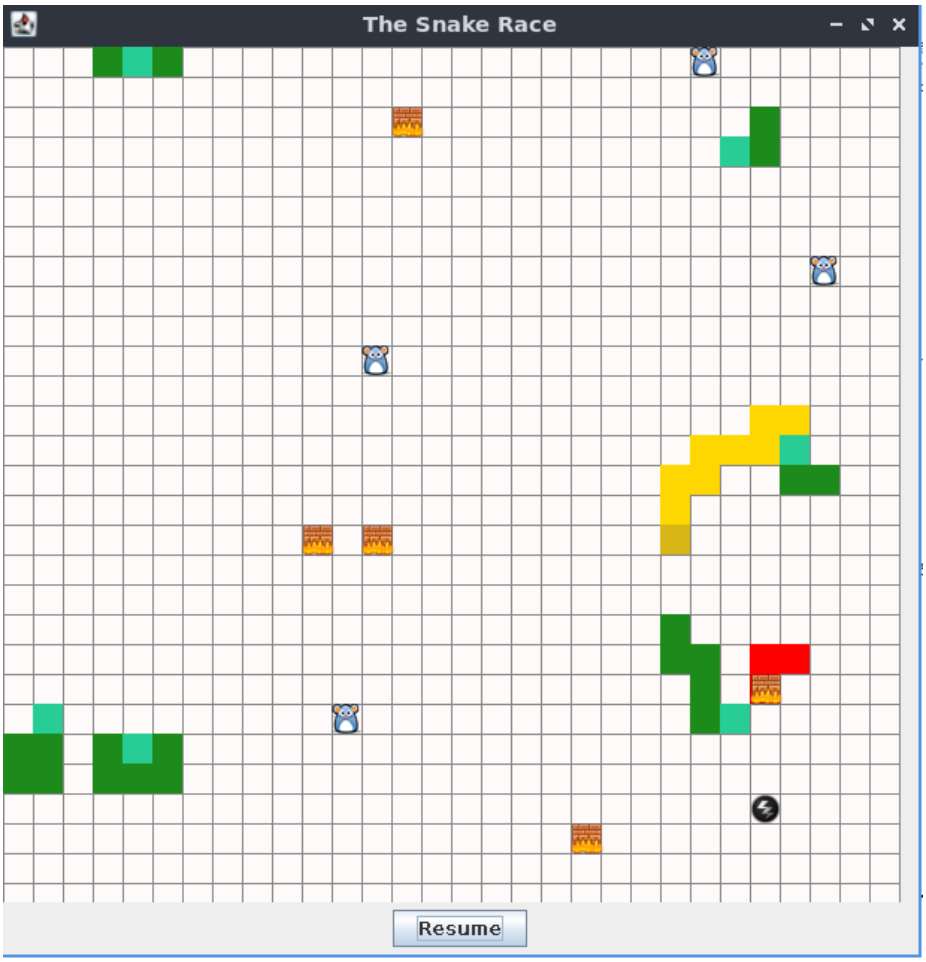
   Puede ser que mientras se está haciendo un  movimiento de una serpiente, se acceda a la información del cuerpo de la serpiente al mismo tiempo que se está añadiendo más partes al cuerpo de esta serpiente, por lo que habría un error por acceso múltiple

   **Uso inadecuado de colecciones:**

   Por estos posibles accesos, es necesario usar algún tipo de colección concurrente (del paquete java.util.concurrent). En nuestro caso usamos LinkedBlockingDeque puesto que era la que más se adaptaba a la necesidad.

   **Esperas activas:**

   Se usa Thread.sleep para generar los fotogramas de las serpientes, pero se podría usar wait en vez, puesto que Thread.sleep no libera el lock en su uso, lo mantiene bloqueado.
3. Se agrega el uso de LinkedBlockingDeque para solucionar las condiciones de carrera.
4. 




En esta imagen se observa a la mejor serpiente en amarillo y a la peor en rojo. Cabe recalcar que, si hay empate en las mejores serpientes, ninguna será selecta como la mejor.

Se adicionan correctamente las opciones de Iniciar, Pausar y Reiniciar.

## CONCLUSIONES

- Avanzamos en el manejo de wait y notify
- Comprendemos más sobre las esperas activas, por qué son un problema y cómo pueden ser solucionadas.
- Se elaboran cambios dentro de la interfaz gráfica satisfactoriamente, aplicando conocimientos previos.
- Se comprende la competencia por recursos, y la necesidad de sincronizar los hilos con el fin de evitar errores.
- Usamos correctamente una implementación de colección concurrente.