**INFORME PRACTICA LABORATORIO**

**LABORATORIO NO. 2 PROGRAMACIÓN CONCURRENTE**

**PRESENTADO A:**

JAVIER IVAN TOQUICA BARRERA

**PRESENTADO POR:**

Juan Sebastián Buitrago Piñeros

Ángel Nicolas Cuervo Naranjo

**ESCUELA COLOMBIANA DE INGENIERIA JULIO GARAVITO**

**ARQUITECTURA DE SOFWARE**

**Grupo 6**

**2025-2**

Tabla de Contenido

[Introducción 3](#_Toc207378428)

[Parte I — (Calentamiento) wait/notify en un programa multi-hilo 3](#_Toc207378429)

[1. Toma el programa PrimeFinder. 3](#_Toc207378430)

[2. Modifícalo para que cada *t* milisegundos: 3](#_Toc207378431)

[3. La sincronización debe usar synchronized, wait(), notify() / notifyAll() sobre el mismo monitor (sin *busy-waiting*). 9](#_Toc207378432)

[4. Reporte de laboratorio las observaciones y/o comentarios explicando tu diseño de sincronización (qué lock, qué condición, cómo evitas *lost wakeups*). 10](#_Toc207378433)

[Conclusiones 11](#_Toc207378434)

[Link Github: 11](#_Toc207378435)

# Introducción

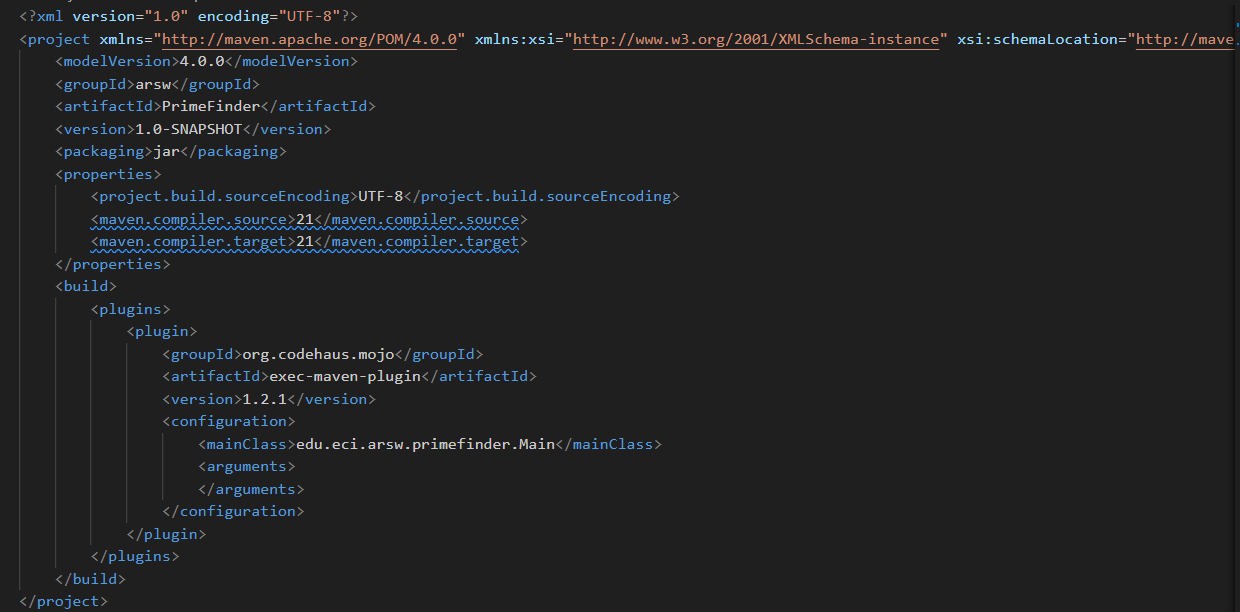
En el presente laboratorio se exploraron conceptos fundamentales de la programación concurrente en Java, haciendo uso de los mecanismos de sincronización provistos por el lenguaje: synchronized, wait() y notify()/notifyAll(). El objetivo principal fue modificar el programa PrimeFinder para coordinar múltiples hilos trabajadores que calculan números primos en paralelo, controlando su ejecución de manera segura y ordenada.

# Parte I — (Calentamiento) wait/notify en un programa multi-hilo

## Toma el programa [PrimeFinder](https://github.com/ARSW-ECI/wait-notify-excercise).

## Modifícalo para que cada *t* milisegundos:

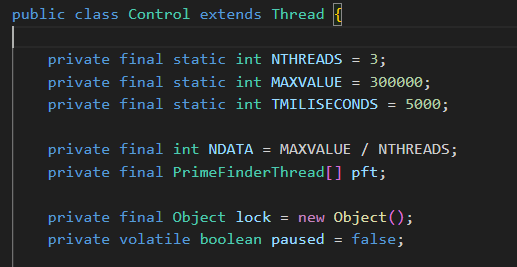
Primero que todo se debe modificar el archivo pom.xml, ya que está en una versión vieja (1.7 -> 21).



* + **Se pausen todos los hilos trabajadores.**

**Clase Control**

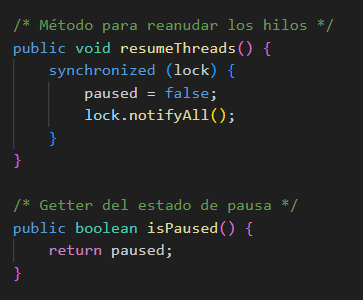
Añadimos un objeto “lock” el cual será el monitor de sincronización compartido y un booleano “paused” que controlará el estado de pausa, este último será “volatile” para que todos los hilos lean directamente de la memoria principal y no del cache.



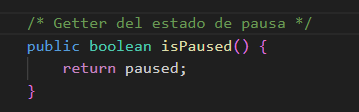
También modificaremos el método run de la clase Control que se encargara de gestiona cuándo los trabajadores se deben pausar y reanudar.



Adicionalmente, creamos el método resumeThreads que se encargará de notificarle a todos los hilos existentes el estado del booleano “paused”.



Por último, un getter del estado del objeto “paused”

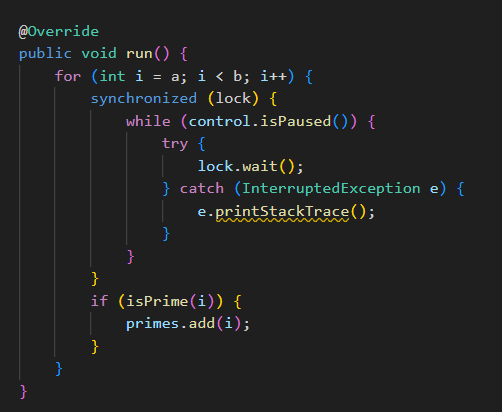


**Clase PrimeFinderThread**

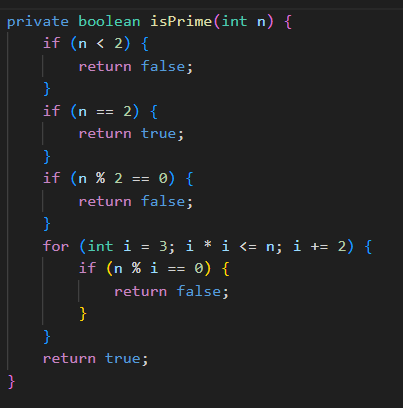
Añadimos el objeto “lock”



Modificamos el método run() para que además de busca primos en su rango en paralelo pueda **sincronizar la pausa/reanudación con el Control** para poder usar wait()/notifyAll() de manera segura.

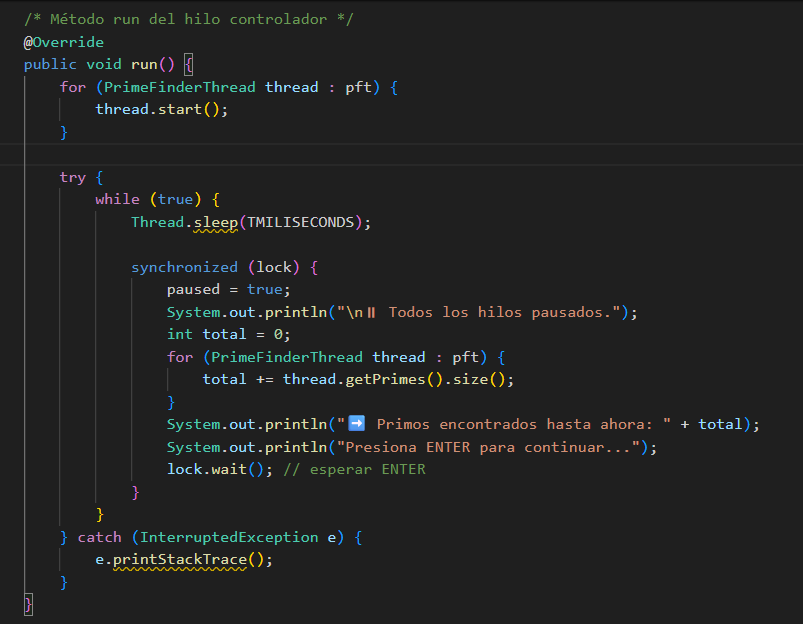


También se modificó el método isPrime() por comodidad



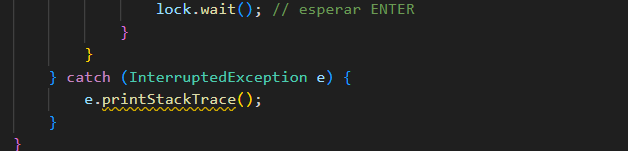
* + **Se muestre cuántos números primos se han encontrado.**

Para esto modificamos el método run() de la clase Control ya que ahí es donde podemos contar los números que lleva cada hilo para luego totalizarlos e imprimirlos.

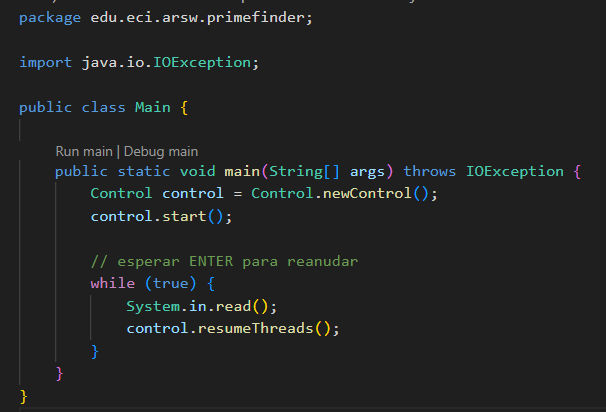
****

* + **El programa espere ENTER para reanudar.**

Para ello modificamos el método run() de la clase Control ejecutando wait() en el objeto “lock” para luego atrapar la excepción “InterruptedException”.

****

También agregamos en la clase main el ciclo while para leer la tecla Enter para luego seguir ejecutando los hilos trabajadores.

****

## La sincronización debe usar synchronized, wait(), notify() / notifyAll() sobre el mismo monitor (sin *busy-waiting*).

* **Uso de synchronized sobre un monitor único**

El objeto lock es el monitor compartido y tanto en PrimeFinderThread.run() como en Control.run() y Control.resumeThreads(), se hace:

* **Uso de wait()**

Cuando los hilos de PrimeFinderThread están en pausa, hacen: lock.wait();

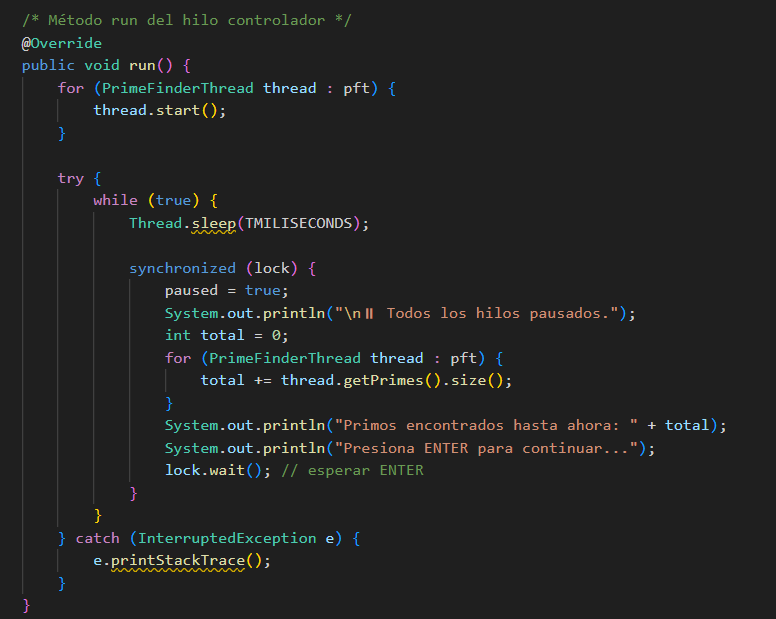
Eso **libera el lock** y suspende al hilo hasta que otro lo despierte. Cabe resaltar que aquí no hay *busy-waiting*, ya que no están haciendo un while(true) revisando la variable paused sin parar.

* **Uso de notifyAll()**

En resumeThreads() se despiertan todos los hilos que estaban en wait():

* **Uso de Mismo monitor**

Tanto synchronized, como wait() y notifyAll() trabajan **siempre sobre el mismo objeto: lock**.

**** ****

## Reporte de laboratorio las observaciones y/o comentarios explicando tu diseño de sincronización (qué lock, qué condición, cómo evitas *lost wakeups*).

Para la sincronización utilicé un objeto lock como monitor compartido entre el hilo controlador y los hilos trabajadores (PrimeFinderThread). Todas las operaciones de pausa y reanudación se realizan dentro de bloques synchronized (lock), garantizando exclusión mutua.

La condición de sincronización es una variable booleana paused, que indica si los hilos deben detener su ejecución temporalmente. Cuando el controlador activa la pausa, entra en el monitor y cambia paused = true. Luego, hace notifyAll() sobre el mismo lock para asegurar que todos los hilos trabajadores reciban la señal de verificar el estado.

Los hilos trabajadores, dentro de sus ciclos de búsqueda de primos, verifican esta condición y si paused == true, ejecutan lock.wait(). Esto los bloquea sin consumir CPU (evitando el busy-waiting) hasta que el controlador cambie paused = false y ejecute nuevamente notifyAll().

Para evitar lost wakeups, siempre la verificación de la condición se hace dentro de un while (paused) y no con if. Así, incluso si un hilo es despertado espuriamente o antes de que paused cambie a false, volverá a esperar correctamente sin avanzar de manera incorrecta.

# Parte II — SnakeRace concurrente (núcleo del laboratorio)

# Conclusiones

El uso de un único objeto lock como monitor permitió centralizar la sincronización entre el hilo controlador y los hilos trabajadores, garantizando exclusión mutua y un flujo de ejecución consistente. La combinación de los métodos wait() y notifyAll() sobre el mismo monitor aseguró que los hilos trabajadores pudieran suspenderse y reanudarse sin desperdiciar ciclos de CPU, evitando el problema del busy-waiting. Además, la condición de control paused, evaluada dentro de un ciclo while, previno los lost wakeups, asegurando que los hilos únicamente reanudaran su trabajo cuando realmente correspondía.

El diseño implementado permitió pausar la ejecución de los hilos cada cierto tiempo, mostrar los resultados parciales y reanudar la búsqueda de primos tras la interacción del usuario, demostrando la correcta integración entre concurrencia y sincronización. Este ejercicio resalta la importancia de diseñar cuidadosamente los mecanismos de coordinación entre hilos, ya que un error en la elección del monitor o en la verificación de condiciones podría derivar en comportamientos incorrectos o ineficiencias dentro del sistema.

Link Github: <https://github.com/elmultiusos/Lab2_SnakeRace-Java21>