Imagen en blanco y negro

Descripción generada automáticamente con confianza baja

Laboratorio 2

Programación concurrente, condiciones de carrera, esquemas de sincronización, colecciones sincronizadas y concurrentes.

Integrantes:

Luisa Fernanda Bermúdez Girón

Karol Daniela Ladino Ladino

Squad:

Inside Out

Profesor:

Javier Iván Toquica Barrera

Curso:

ARSW – 1

Fecha De Entrega:

01-09-2023

1. **Introducción**

En este laboratorio, exploraremos conceptos fundamentales de programación concurrente, condiciones de carrera y esquemas de sincronización. Estudiaremos cómo manejar múltiples hilos de ejecución de manera eficiente y segura en entornos paralelos.

A través de ejercicios prácticos, nos adentraremos en el uso de herramientas cruciales como wait y notify, perfeccionaremos nuestras habilidades para identificar y resolver condiciones de carrera, y aplicaremos estos conocimientos al enriquecimiento y optimización de una aplicación de juego interactiva.

1. **Desarrollo del laboratorio**

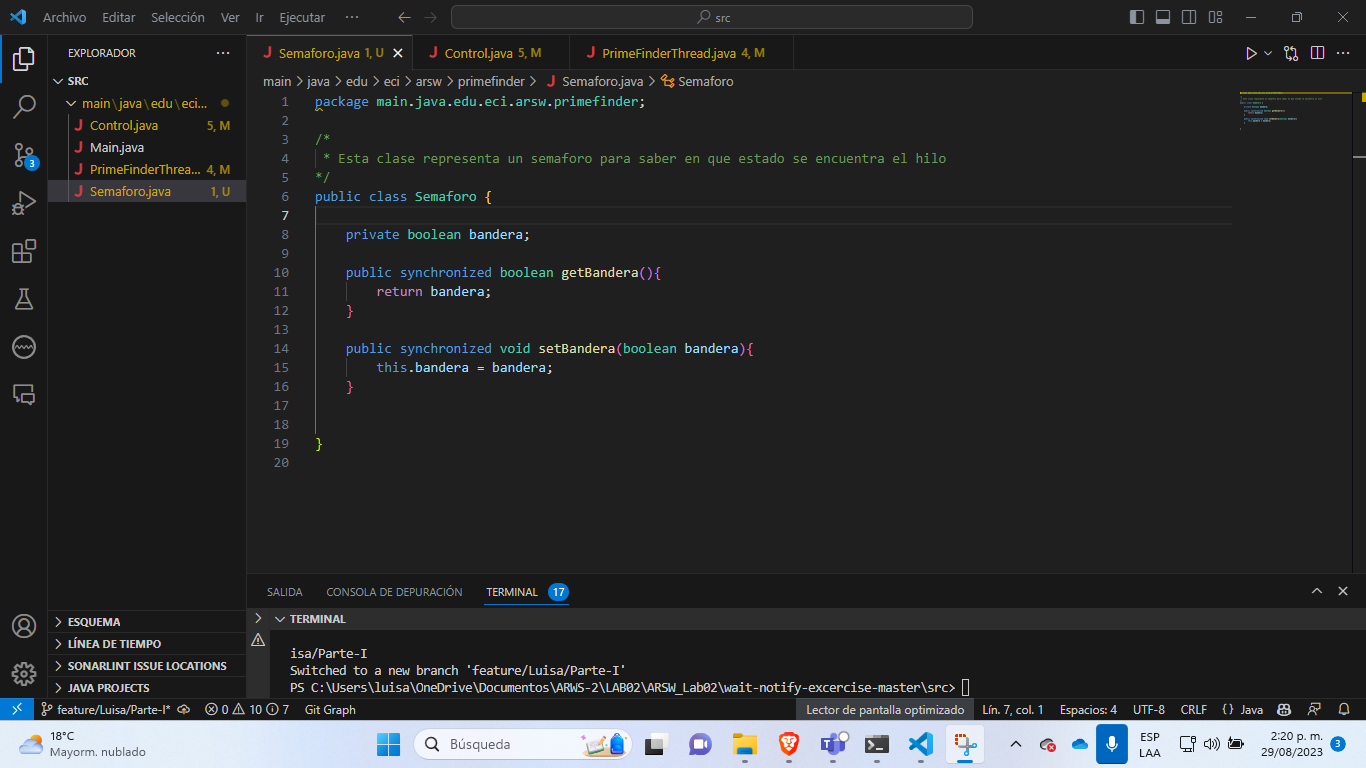
**Parte I**

Control de hilos con [wait/notify.](http://howtodoinjava.com/core-java/multi-threading/how-to-work-with-wait-notify-and-notifyall-in-java/)

1. Descargue el proyecto [PrimeFinder](https://github.com/ARSW-ECI/wait-notify-excercise). Este es un programa que calcula números primos entre 0 y M (Control.MAXVALUE), concurrentemente, distribuyendo la búsqueda de los mismos entre n (Control.NTHREADS) hilos independientes.
2. 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 los mismos. Utilice los mecanismos de sincronización provistos por el lenguaje (wait y notify, notifyAll).

Tenga en cuenta:

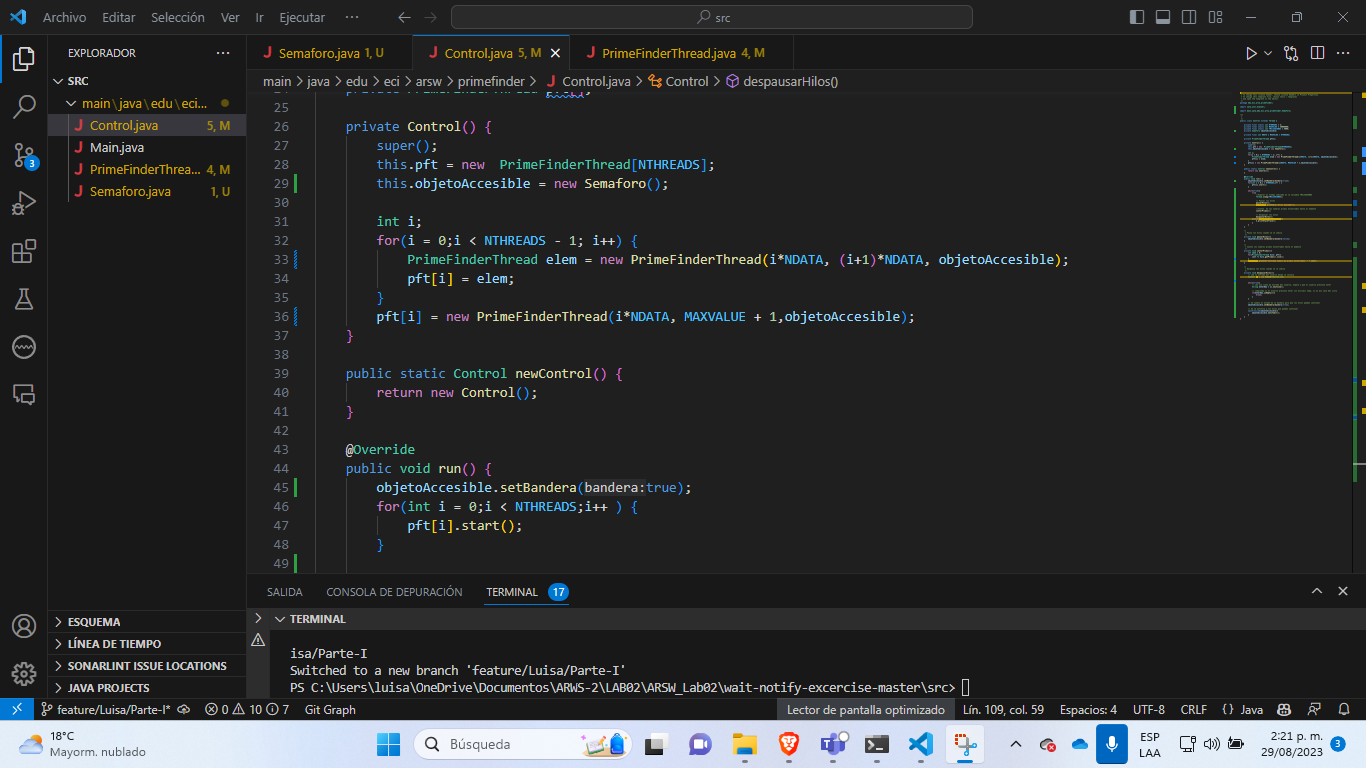
* La construcción synchronized se utiliza para obtener acceso exclusivo a un objeto.
* La instrucción A.wait() ejecutada en un hilo B pone en modo suspendido al hilo B (independientemente de qué objeto 'A' sea usado como 'lock'). Para reanudarlo, otro hilo activo puede reanudar a B haciendo 'notify()' al objeto usado como 'lock' (es decir, A).
* La instrucción notify(), despierta el primer hilo que hizo wait() sobre el objeto.
* La instrucción notifyAll(), despierta todos los hilos que estan esperando por el objeto (hicieron wait()sobre el objeto).
* Creamos la clase Semaforo la cual nos permitirá conocer en que estado se encuentra el hilo



* Modificación de la clase Control

Captura de pantalla de computadora

Descripción generada automáticamente



Captura de pantalla de computadora

Descripción generada automáticamente

Captura de pantalla de computadora

Descripción generada automáticamente

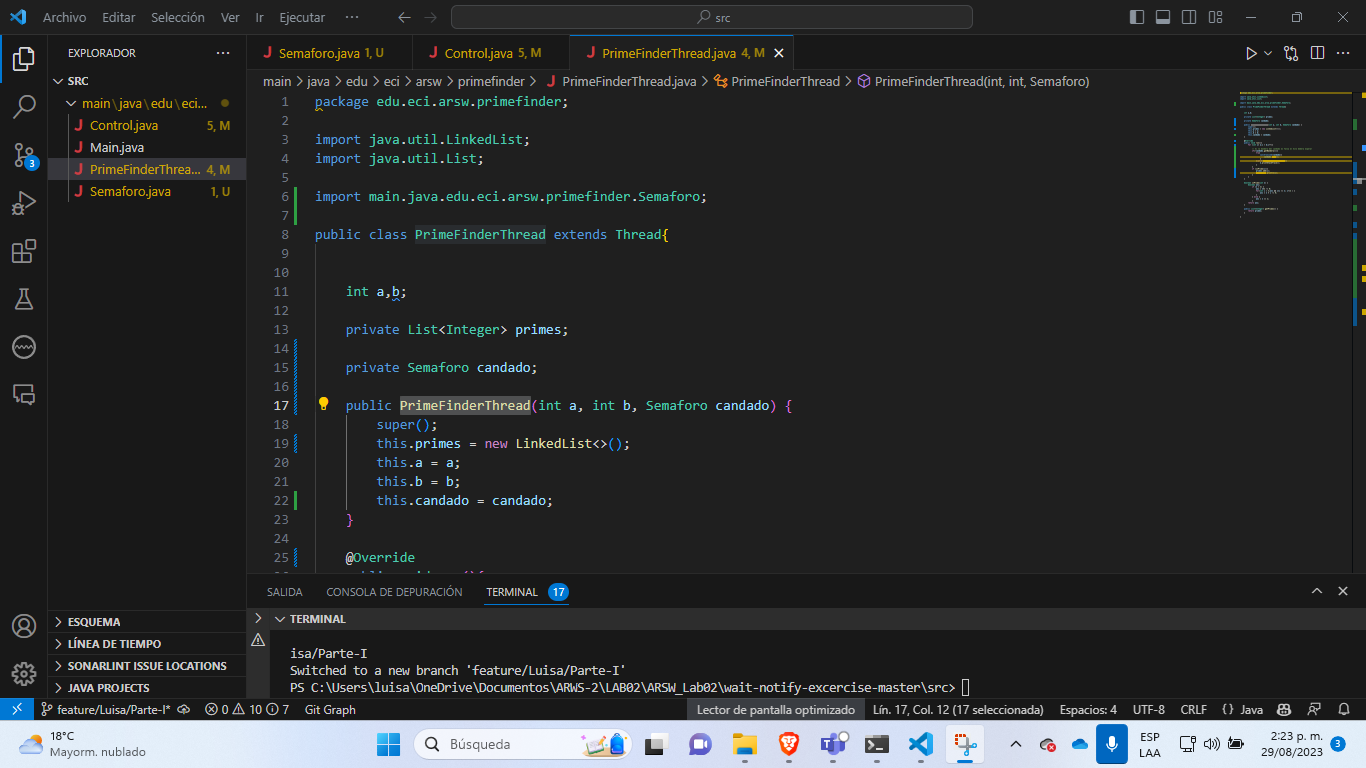
Texto

Descripción generada automáticamente

Captura de pantalla de computadora

Descripción generada automáticamente

* Modificación de la clase PrimeFinderThread



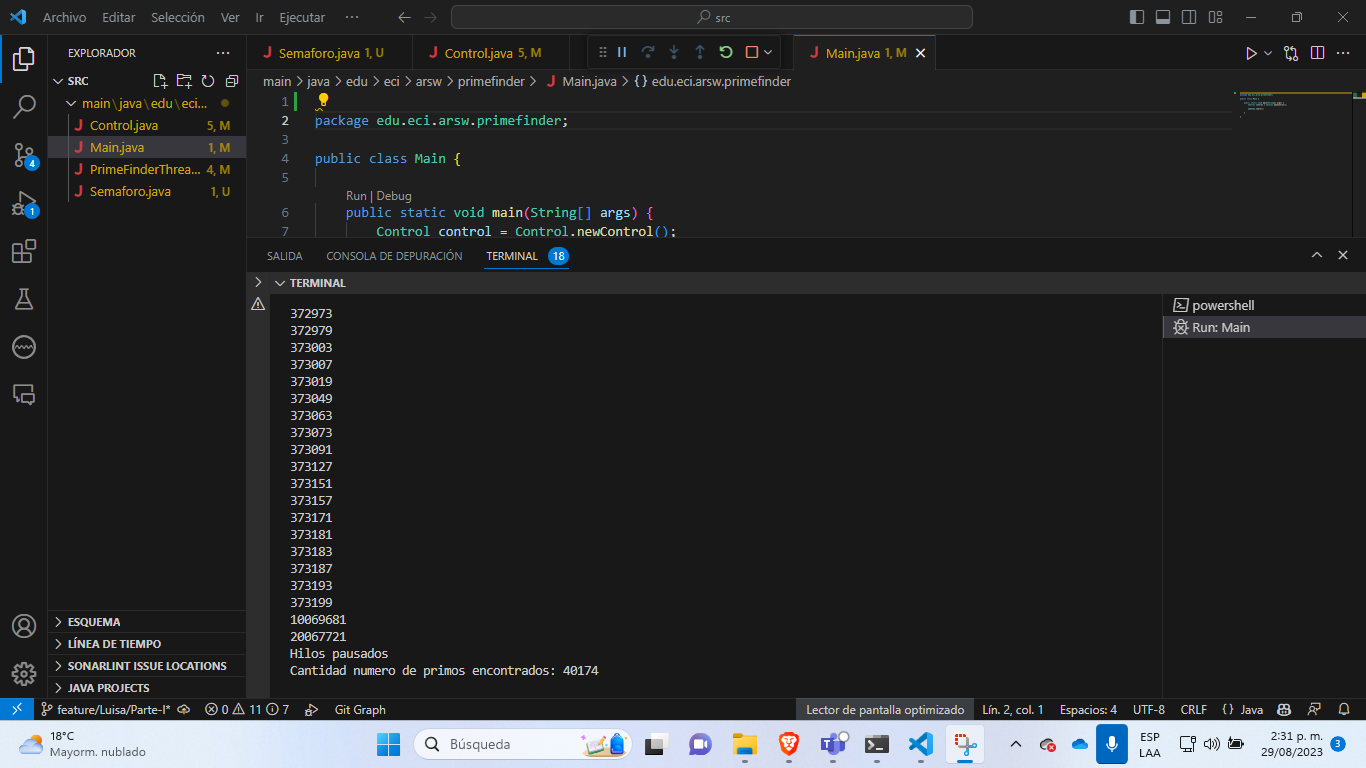
Captura de pantalla de computadora

Descripción generada automáticamente

Captura de pantalla de computadora

Descripción generada automáticamente

* Resultados ejecución:



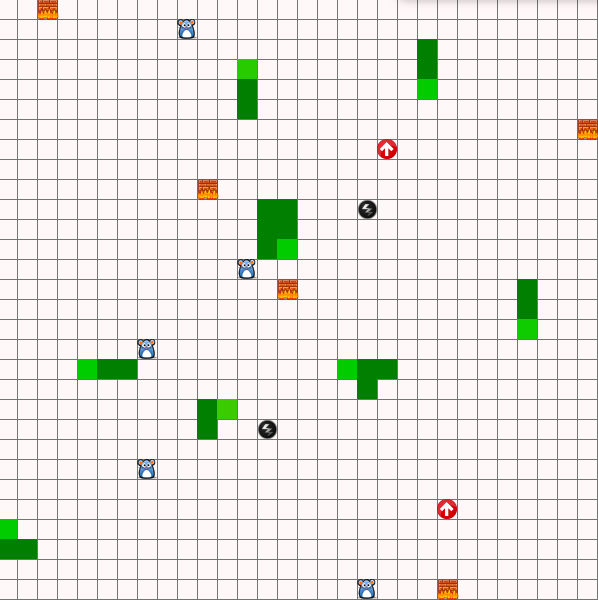
Captura de pantalla de computadora

Descripción generada automáticamente

**Parte II**

SnakeRace es una versión autónoma, multi-serpiente del famoso juego 'snake', basado en el proyecto de João Andrade -este ejercicio es un 'fork' del mismo-. En este juego:

* N serpientes funcionan de manera autónoma.
* No existe el concepto de colisión entre las mismas. La única forma de que mueran es estrellándose contra un muro.
* Hay ratones distribuídos a lo largo del juego. Como en el juego clásico, cada vez que una serpiente se come a un ratón, ésta crece.
* Existen unos puntos (flechas rojas) que teletransportan a las serpientes.
* Los rayos hacen que la serpiente aumente su velocidad.

[](https://github.com/ARSW2023-2/ARSW_Lab02/blob/main/img/sshot.png)

**Ejercicio**

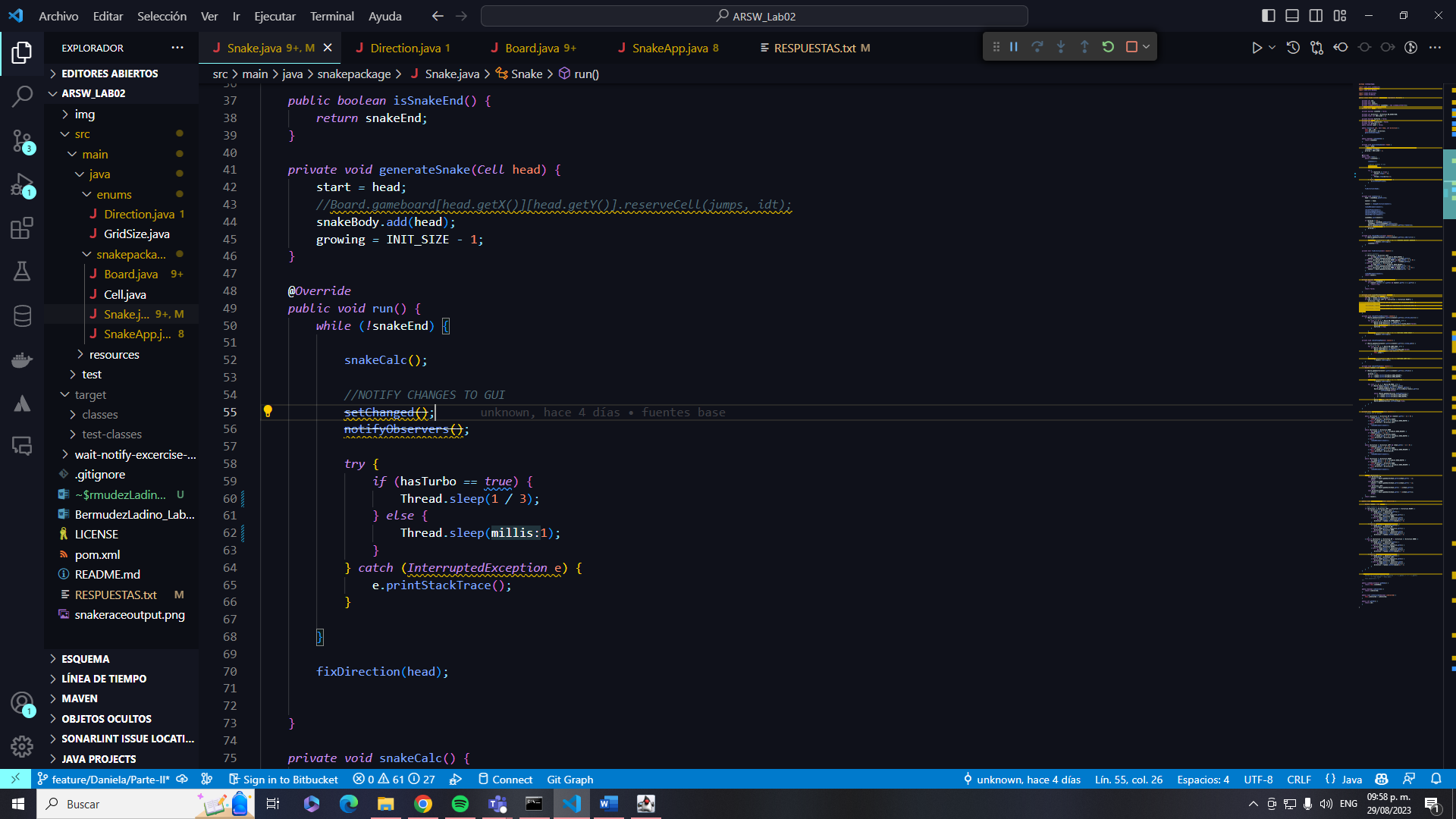
1. Analice el código para entender cómo hace uso de hilos para crear un comportamiento autónomo de las N serpientes.

La clase Snake implementa la interfaz Runnable, lo que significa que cada instancia de serpiente puede ejecutarse en su propio hilo el cual ejecuta la lógica de movimiento y comportamiento de cada una de las serpientes.

SnakeApp crea instancias de serpientes y hilos para cada serpiente, permitiendo que todas las serpientes se muevan y actúen autónomamente en paralelo.

1. De acuerdo con lo anterior, y con la lógica del juego, identifique y escriba claramente (archivo RESPUESTAS.txt):
   * Posibles condiciones de carrera.
2. Revisar si una celda esta libre o no.
3. Si hay comida disponible en una casilla.
4. Si una flecha roja está disponible o no.
5. Si un rayo está disponible o no.
   * Uso inadecuado de colecciones, considerando su manejo concurrente (para esto, aumente la velocidad del juego y ejecútelo varias veces hasta que se genere un error).

Dentro de la clase Snake, ajustamos los valores de los tiempos de espera en el método run(). Para esto reducimos los valores cambiando de 500 milisegundos a 1 milisegundo dentro del método Thread.sleep() para que las serpientes se movieran más rápido.



Como podemos ver en la siguiente imagen luego de ejecutar el juego el error que sale es ConcurrentModificationException, este error se puede dar en el momento en que se está dibujando la serpiente ya que la referencia que pide el tablero para dibujarla se está iterando mientras la serpiente se modifica (agregando o eliminando elementos).

Captura de pantalla de computadora

Descripción generada automáticamente

* + Uso innecesario de esperas activas.

La espera innecesaria está en la clase SnakeApp en el while del metodo init, ya que este bucle verifica constantemente si todas las serpientes han llegado al final del juego. Cuando todas las serpientes hayan terminado, el bucle se rompe.

1. Identifique las regiones críticas asociadas a las condiciones de carrera, y haga algo para eliminarlas. Tenga en cuenta que se debe sincronizar estrictamente LO NECESARIO. En su documento de respuestas indique, la solución realizada para cada ítem del punto 2. Igualmente tenga en cuenta que en los siguientes puntos NO se deben agregar más posibles condiciones de carrera.
2. Como se puede observar, el juego está incompleto. Haga los ajustes necesarios para que a través de botones en la interfaz se pueda Iniciar/Pausar/Reanudar el juego: iniciar el juego no se ha iniciado aún, suspender el juego si está en ejecución, reactivar el juego si está suspendido. Para esto tenga en cuenta:
   * Al pausar (suspender) el juego, en alguna parte de la interfaz (agregue los componentes que desee) se debe mostrar:
     + La serpiente viva más larga.
     + La peor serpiente: la que primero murió.

Recuerde que la suspensión de las serpientes NO es instantánea, y que se debe garantizar que se muestre información consistente.

1. **Conclusiones**