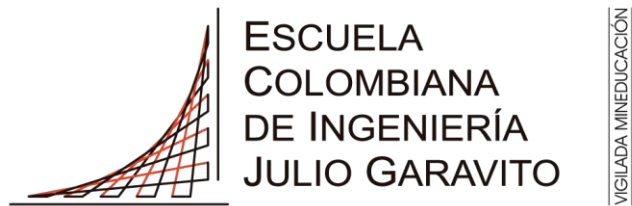


Escuela Colombiana de Ingeniería Julio Garavito



UNIVERSIDAD

LABORATORIO No.2

Presentado por:

Esteban Aguilera Contreras

Carlos David Barrero Velásquez

Juan Sebastián Beltrán Rodríguez

Iván Santiago Forero Torres

Asignatura:

ISIS – ARSW (Redes de computadores)

Grupo:

Grupo 5

Docente:

Javier Ivan Toquica Barrera

29/08/2025

TABLA DE CONTENIDO

OBJETIVOS.....	3
Objetivo General:	3
Objetivos Específicos:	3
Herramientas Utilizadas:	3
Requisitos:.....	3
MARCO TEORICO.....	4
EJECUCIÓN DEL LABORATORIO	6
Parte 1: Wait and Notify	6
Parte 2: SnakeRace concurrente (núcleo del laboratorio)	9
CONCLUSIONES.....	31
WEBGRAFIA.....	32

OBJETIVOS

Objetivo General:

- Diseñar, implementar y analizar un sistema concurrente en Java 21 mediante el juego *Snake Race*, aplicando *virtual threads*, sincronización de procesos y colecciones seguras, con el fin de comprender y resolver problemas de condiciones de carrera, garantizar la correcta coordinación de hilos y fortalecer la robustez y consistencia de aplicaciones concurrentes bajo distintas cargas de ejecución.

Objetivos Específicos:

- Comprender y aplicar los fundamentos de la programación concurrente en Java, explorando el uso de *virtual threads* y los mecanismos de sincronización tradicionales (*synchronized*, *wait*, *notify*, *notifyAll*).
- Identificar y analizar condiciones de carrera, esperas activas y uso indebido de colecciones no seguras, documentando su impacto en la consistencia de los datos y el comportamiento del sistema.
- Implementar soluciones a los problemas de concurrencia, minimizando regiones críticas y sustituyendo esperas activas por mecanismos eficientes de coordinación como monitores, señales o utilidades de la librería de concurrencia de Java.
- **Evaluar la robustez del sistema bajo alta carga de trabajo**, garantizando que el juego funcione correctamente con múltiples serpientes y velocidades elevadas sin producir excepciones de concurrencia ni bloqueos.

Herramientas Utilizadas:

- Visual Estudio Code.
- Maven.
- Git/Github (trabajo colaborativo)
- .
- .

Requisitos:

- JDK 21.
- Maven 3.9
- SO: Windows, macOS o Linux.

MARCO TEORICO

La programación concurrente consiste en la ejecución simultánea de múltiples tareas que pueden interactuar entre sí. Su propósito es mejorar la eficiencia, reducir tiempos de respuesta y aprovechar las arquitecturas modernas de procesadores multinúcleo. En Java, la concurrencia es un componente esencial soportado desde sus primeras versiones, pero ha evolucionado con nuevas herramientas como los *virtual threads* en Java 21, que permiten un mayor grado de paralelismo con menor consumo de recursos.

Un hilo es la unidad básica de ejecución dentro de un proceso. Los platform threads (tradicionales) se limitan en número debido al costo de recursos del sistema operativo. Con Java 21 surge el concepto de virtual threads, una implementación liviana de hilos gestionados por la JVM que permite manejar miles de hilos concurrentes sin el alto costo de los hilos del sistema operativo.

- Ventajas de los virtual threads:
 - Bajo consumo de memoria.
 - Creación y gestión más rápidas.
 - Ideal para aplicaciones con gran número de tareas concurrentes, como el juego *Snake Race*.

Una condición de carrera ocurre cuando dos o más hilos acceden y modifican un recurso compartido sin la debida sincronización, provocando resultados inconsistentes o impredecibles. En el contexto del laboratorio, estas condiciones pueden darse cuando varias serpientes comparten estructuras de datos como el tablero, posiciones o listas de elementos.

Para evitar las condiciones de carrera, se requiere sincronización:

- `synchronized`: bloquea el acceso concurrente a un recurso compartido.
- `wait()` y `notify()/notifyAll()`: permiten la comunicación entre hilos a través de un monitor, evitando la espera activa (*busy-waiting*). El uso eficiente de monitores garantiza que los hilos cooperen sin desperdiciar recursos, aspecto central en la Parte I del laboratorio.

Las colecciones estándar como `ArrayList` o `HashMap` no son seguras en entornos concurrentes. Java provee alternativas como:

- `ConcurrentHashMap`

- CopyOnWriteArrayList

- BlockingQueue

Estas estructuras permiten acceso concurrente seguro, evitando excepciones como `ConcurrentModificationException` que se pueden presentar al ejecutar con múltiples serpientes.

La espera activa (busy-waiting) consiste en que un hilo revisa constantemente una condición, consumiendo CPU innecesariamente. La buena práctica es sustituirla por mecanismos de señalización (wait/notify, Locks, Condition) o por clases utilitarias (Semaphore, CountDownLatch, CyclicBarrier) que permiten sincronización más eficiente.

En un sistema concurrente, pausar o reanudar implica coordinar múltiples hilos para evitar estados intermedios inconsistentes. El GameClock y los controles de UI requieren garantizar atomicidad en operaciones como “pausar y mostrar estado” para evitar *tearing* (inconsistencias visuales).

Un sistema concurrente debe mantenerse estable aún bajo condiciones extremas de carga. La robustez implica:

- Evitar bloqueos mutuos (deadlocks).
- Controlar condiciones de carrera.
- Manejar grandes volúmenes de hilos sin degradación significativa. En este laboratorio, la robustez se evalúa al correr con más de 20 serpientes o con alta velocidad, garantizando estabilidad en la simulación.

EJECUCIÓN DEL LABORATORIO

Parte 1: Wait and Notify

1. Lock (Monitor)

- Para controlar las acciones de pause and resume de los hilos, se definió un objeto monitor dentro de la clase control

```
40     private final Object monitor = new Object();
41     private boolean paused = false;
42
```

- El monitor garantiza que solo un hilo pueda entrar a la sección critica donde se cambia o se consulta paused
- Todas las operaciones relacionadas con pausa and resume se hacen dentro del bloque de synchronized

```
public void pauseAll() {
    synchronized (monitor){
        paused = true;
    }
}
```

```
129
130     public void resumeAll() {
131         synchronized (monitor){
132             paused = false;
133             monitor.notifyAll();
134         }
135     }
136
```

```
public boolean isPaused() {
    synchronized (monitor) {
        return paused;
    }
}
```

- Esto asegura que no haya data races al leer o modificar paused.
- Además, el alcance del monitor es mínimo ya que solo protege la consulta o el cambio de estado , no el cálculo de los primos, para no bloquear el procesamiento de manera innecesaria

2. Condición de espera (wait/notifyAll) y prevenir Lost WakeUps

- Los hilos trabajadores (PrimeFinderThread) consultan continuamente si deben detenerse
- Cuando el Control marca paused=true, los hilos se bloquean con wait() sobre el monitor
- Para prevenir los Lost WakeUps es decir, que un hilo se despierte sin cumplir la condición o se perdió la notificación, usamos el patrón while accediendo a la condición de pausa del control asegurando que si un hilo recibe un notifyAll() de forma anticipada, al volver a comprobar esta condición seguirá esperando si todavía esta en pausa

```
30  @Override
31  public void run(){
32      for (int i= a;i < b;i++){
33          synchronized (control.getMonitor()) {
34              while(control.isPaused()) {
35                  try {
36                      control.getMonitor().wait();
37                  } catch (InterruptedException ex) {}
38                  System.out.println(x:"Thread interrupted");
39              }
40          }
41      }
42  }
```

- Posteriormente si el usuario presiona Enter, el control llama al método ResumeAll() generando que todos los hilos que estaban detenidos en wait(), reciban la señal y continúen con su ejecución
- Esto elimina la necesidad de una espera activa, ya que los hilos quedan bloqueados en wait() y no consumen CPU hasta recibir notifyAll().

3. Interacción con el usuario

Temporizador

- En el hilo principal se usa un bucle con Thread.sleep(TMILISECONDS) para esperar el tiempo configurado
- Una vez se transcurra el tiempo, se ejecuta pauseAll() y bloquea todos los hilos trabajadores

```
//sleep the principal thread
Thread.sleep(TMILISECONDS);
//sleep the list thhe PrimeFinderThreads
pauseAll();
```

Mostrar cantidad de números primos encontrados

- En la clase Control, después de pausar todos los hilos , se recorre el arreglo que los contiene y se suma el tamaño de las listas de primos de cada hilo
- Esta operación no interfiere con la sincronización, porque la lista de primos se modifica dentro de cada hilo y nunca se comparte entre ellos

```
//Count Primes
int totalPrimes=0;
for (PrimeFinderThread thread : pft) {
    totalPrimes += thread.getPrimes().size();
}
System.out.println("Total primes found: " + totalPrimes);
```

Esperar ENTER para reanudar

- Luego de mostrar las estadísticas , el hilo principal espera que el usuario presione ENTER antes de reanudar los hilos
- Este paso asegura que el usuario tenga tiempo para observar las estadísticas
- Esto se implementa con la siguiente instrucción donde:
 - i. Readline() captura la entrada del usuario
 - ii. isEmpty() valida que solo haya presionado ENTER
 - iii. La condición de while obliga a repetir el mensaje de error hasta que presione la tecla ENTER

```
//only pass if the user press Enter. Readline returns "" if is enter
while (!(input = System.console().readLine()).isEmpty()){
    System.out.println(x:"Only Enter is valid. Try again");
}

resumeAll();
```

- Una vez se presione ENTER, el bucle se rompe y Control llama a resumeAll() generando que todos los hilos bloqueados, continúen con el calculo de los primos

Parte 2: SnakeRace concurrente (núcleo del laboratorio)

1. Análisis de concurrencia:

- Explica cómo el código usa hilos para dar autonomía a cada serpiente:
- La clase *SnakeRunner* ubicada en la carpeta *concurrency* implementa la interfaz “*Runnable*”, generando que cada instancia de Snake sea un hilo, al crear un objeto *SnakeRunner* se reescribe el método *run()* el cual contiene la ejecución de la serpiente como moverse, verificar colisiones y demás, permitiendo que cada serpiente tenga autonomía propia y no se ve afectada por las demás.

```
@Override
public void run() {
    try {
        while (!Thread.currentThread().isInterrupted()) {
            maybeTurn();
            var res = board.step(snake);
            if (res == Board.MoveResult.HIT_OBSTACLE) {
                randomTurn();
            } else if (res == Board.MoveResult.ATE_TURBO) {
                turboTicks = 100;
            }
            int sleep = (turboTicks > 0) ? turboSleepMs : baseSleepMs;
            if (turboTicks > 0) turboTicks--;
            Thread.sleep(sleep);
        }
    } catch (InterruptedException ie) {
        Thread.currentThread().interrupt();
    }
}

private void maybeTurn() {
    double p = (turboTicks > 0) ? 0.05 : 0.10;
    if (ThreadLocalRandom.current().nextDouble() < p) randomTurn();
}

private void randomTurn() {
    var dirs = Direction.values();
    snake.turn(dirs[ThreadLocalRandom.current().nextInt(dirs.length)]);
}
}
```

- También se evidencia en la clase *SnakeApp* la autonomía de cada serpiente a través de:

```
var exec = Executors.newVirtualThreadPerTaskExecutor();
snakes.forEach(s -> exec.submit(new SnakeRunner(s, board)));
```

- Posible Condiciones de carrera (dos o más hilos acceden y modifican datos compartidos al mismo tiempo):
- En la clase Board tenemos colecciones compartidas tales como mice, obstacles, turbo y teletransports donde los hilos de las serpientes pueden interactuar con ellos, si dos serpientes intentan comer al mismo ratón en el mismo momento se puede presentar una acción imprecisa alterando los resultados.

```
// lista de ratones que se comen las serpientes
private final Set<Position> mice = new HashSet<>();
// lista de obstaculos, muere si choca
private final Set<Position> obstacles = new HashSet<>();
// lista de turbos
private final Set<Position> turbo = new HashSet<>();
// lista de portales
private final Map<Position, Position> teleports = new HashMap<>();
```

- En la clase Snake se utiliza volatile para la dirección generando que se guarde una copia local en el cache de cada objeto, si el jugador cambia la dirección y el hilo de movimiento se ejecuta al mismo tiempo puede generar un resultado no esperado para el jugador.

```
// guarda la direccion de la serpiente
private volatile Direction direction;
```

- La interfaz de usuario puede llegar a presentar condiciones de carrera, si un hilo llega a modificar los valores mientras el método paintComponent lo dibuja puede llegar a presentar inconsistencias.
- En el reloj si lo llegamos a pausar en medio de un cambio de tablero o de comportamiento se pueden presentar cambios o comportamientos no deseados.
- Colecciones o estructuras no seguras en contexto concurrente.
- Las estructuras HashSet y HashMap por defecto no son seguras, más sin embargo están protegidas en el código con la palabra synchronized asiando que sean accesibles por un hilo a la vez.

```
//funciones que retornan las posiciones de los elementos en el tablero
// se utiliza synchronized asegurar que un solo hilo acceda a la vez evitando errores
public synchronized Set<Position> mice() { return new HashSet<>(mice); }
public synchronized Set<Position> obstacles() { return new HashSet<>(obstacles); }
public synchronized Set<Position> turbo() { return new HashSet<>(turbo); }
public synchronized Map<Position, Position> teleports() { return new HashMap<>(teleports); }
```

- Las ArrayList no son seguras por defecto, a diferencia del anterior caso no se está utilizando synchronized, si dos hilos acceden al mismo tiempo y cada uno realiza una modificación puede tener comportamientos imprecisos o lanzar la excepción ConcurrentModificationException que se encuentra por default en la importación java.util.*

```
// deque es Double ended Queue guarda las posiciones de la serpiente
private final Deque<Position> body = new ArrayDeque<>();
```

- ArrayList es una lista dinámica en Java, al igual que en el caso anterior no se utilizan synchronized, aunque no representa un riesgo real puesto que no se agregan ni eliminan serpientes .

```
private final java.util.List<Snake> snakes = new java.util.ArrayList<>();
```

- Ocurrencias de espera activa (busy-wait) o de sincronización innecesaria.
- Busy-wait ocurre cuando un hilo accede de manera contante a una condición consumiendo recursos como CPU esperando que haya algún cambio en el comportamiento, en la clase SnakeRunner podemos ver que hay un bucle, pero este al final descansa el hilo por esta razón no hay espera activa.

```
@Override
public void run() {
    try {
        //mientras que el thread actual no este interrumpido
        while (!Thread.currentThread().isInterrupted()) {
            //llama a una funcion mas abajo
            maybeTurn();
            //ejecuta un paso en el tablero con la serpiente
            var res = board.step(snake);
            //si la serpiente choca con un obstaculo
            if (res == Board.MoveResult.HIT_OBSTACLE) {
                // giro aleatorio para evitarlo
                randomTurn();
            }
            // si se comio un turbo
            else if (res == Board.MoveResult.ATE_TURBO) {
                // activa 100 iteraciones
                turboTicks = 100;
            }

            // determina el tiempo de espera
            int sleep = (turboTicks > 0) ? turboSleepMs : baseSleepMs;
            // si el turbo esta activo elimina el contador en 1 para la duracion restante
            if (turboTicks > 0) turboTicks--;
            // pausa el hilo
            Thread.sleep(sleep);
        }
    } catch (InterruptedException ie) {
        Thread.currentThread().interrupt();
    }
}
```

- La sincronización innecesaria ocurre cuando se utiliza synchronized y locks sin que haya riesgo real de acceso concurrente, a lo largo del código si se evidenciaron varios synchronized pero son necesarios para proteger los datos y no se presencié ningún tipo de locks.

2. Correcciones mínimas y regiones críticas:

- Para el siguiente punto se incorporó la siguiente modalidad:
 1. Nombre.
 2. Prioridad para el problema encontrado, la cual se clasifica en cuatro categorías (ALTA, CRITICA, MEDIA y BAJA).

3. Descripción del riesgo.
 4. Solución implementada.
 5. Justificación.
- Eliminación de Esperas Activas en SnakeRunner
 - Prioridad:** ALTA
 - Riesgo Original:**
 - Thread.sleep() causaba esperas activas desperdiciando CPU.
 - Falta de coordinación con GameClock generaba movimientos desincronizados.
 - Control de pause/resume inconsistente entre serpientes.
 - Solución Implementada:**
 - Reemplazado Thread.sleep() por mecanismo wait/notify.
 - Implementada interfaz GameClockListener para coordinación.
 - Agregar métodos pause(), resume(), stop() con sincronización.
 - Por qué esta solución:**
 - wait/notify libera CPU durante esperas (más eficiente).
 - Coordinación centralizada con GameClock evita desincronización.
 - Control unificado de estado permite pause/resume instantáneo.
 - Sistema híbrido con fallback timing para robustez.

Con wait/notify:

```
@Override
public void run() {
    try {
        lastMoveTime = System.currentTimeMillis();
        while (!Thread.currentThread().isInterrupted() && !stopped) {
            synchronized (pauseLock) {
                while (paused && !stopped) {
                    pauseLock.wait();
                }
            }

            if (stopped)
                break;

            synchronized (pauseLock) {
                while (!clockTick && !stopped) {
                    int sleepTime = (turboTicks > 0) ? turboSleepMs : baseSleepMs;
                    pauseLock.wait(sleepTime);

                    if (!clockTick) {
                        long currentTime = System.currentTimeMillis();
                        if (currentTime - lastMoveTime >= sleepTime) {
                            break;
                        }
                    }
                }
            }
            clockTick = false;
        }

        if (stopped)
            break;

        lastMoveTime = System.currentTimeMillis();

        maybeTurn();
        var res = board.step(snake);
        if (res == Board.MoveResult.HIT_OBSTACLE) {
            randomTurn();
        } else if (res == Board.MoveResult.ATE_TURBO) {
            turboTicks = 100;
        }
        if (turboTicks > 0)
            turboTicks--;
    }
    catch (InterruptedException ie) {
        Thread.currentThread().interrupt();
    }
}
```

- Métodos pause/resume/stop:

```
/**
 * Pauses the snake runner using wait/notify mechanism
 */
public void pause() {
    synchronized (pauseLock) {
        paused = true;
    }
}
```

```
/**
 * Resumes the snake runner using wait/notify mechanism
 */
public void resume() {
    synchronized (pauseLock) {
        paused = false;
        pauseLock.notifyAll();
    }
}
```

```
/**
 * Stops the snake runner
 */
public void stop() {
    synchronized (pauseLock) {
        stopped = true;
        pauseLock.notifyAll();
    }
}
```

```

/**
 * Triggers an immediate move by notifying waiting threads
 */
public void tick() {
    synchronized (pauseLock) {
        if (!paused && !stopped) {
            pauseLock.notify();
        }
    }
}

```

- Coordinación con GameClock:

```

@Override
public void onTick() {
    synchronized (pauseLock) {
        clockTick = true;
        pauseLock.notify();
    }
}

```

```

@Override
public void onPause() {
    pause();
}

```

```

@Override
public void onResume() {
    resume();
}


```

```

@Override
public void onStop() {
    stop();
}

```

- Para coordinación:



```

@FunctionalInterface
public interface GameClockListener {
    void onTick();

    default void onPause() {
    }

    default void onResume() {
    }

    default void onStop() {
    }
}

```

- Lista de listeners y notificación:

```

- private final List<GameClockListener> listeners = new CopyOnWriteArrayList<>();
-
- public void addListener(GameClockListener listener) {
-     listeners.add(listener);
- }
-
- public void pause() {
-     state.set(GameState.PAUSED);
-     listeners.forEach(GameClockListener::onPause);
- }
-
- public void resume() {
-     state.set(GameState.RUNNING);
-     listeners.forEach(GameClockListener::onResume);
- }
-
- private void notifyListeners() {
-     listeners.forEach(GameClockListener::onTick);
- }

```

Colecciones Thread-Safe en Board

Prioridad: CRITICA

Riesgo Original:

- HashSet *mice* causaba ConcurrentModificationException.
- HashSet *obstacles* generaba inconsistencias en lecturas.
- HashSet *turbo* tenía accesos no atómicos.
- HashMap<Position, Position> *teleports* vulnerable a corrupción.

Solución Implementada:

- HashSet → *ConcurrentHashMap.newKeySet()* para *mice*, *obstacles*, *turbo*.
- HashMap → *ConcurrentHashMap* para *teleports*.
- Eliminados *synchronized* en métodos de acceso (*mice()*, *obstacles()*, etc.).

Por qué esta solución:

- Thread-safe sin *locks* explícitos mejora el rendimiento.
- Operaciones atómicas previenen *ConcurrentModificationException*.
- Accesos concurrentes seguros sin bloquear otros hilos.
- Mejor escalabilidad con mayor número de serpientes.

▪ Colecciones thread-safe:

```
- package co.eci.snake.core;
-
- import java.util.Map;
- import java.util.Objects;
- import java.util.Set;
- import java.util.concurrent.ThreadLocalRandom;
- import java.util.concurrent.ConcurrentHashMap;
- import java.util.concurrent.locks.ReentrantLock;
-
- private final Set<Position> mice = ConcurrentHashMap.newKeySet();
- private final Set<Position> obstacles = ConcurrentHashMap.newKeySet();
- private final Set<Position> turbo = ConcurrentHashMap.newKeySet();
- private final Map<Position, Position> teleports = new ConcurrentHashMap<>();
- private final ReentrantLock miceLock = new ReentrantLock();
- private final ReentrantLock itemGenerationLock = new ReentrantLock();
```

▪ Acceso directo thread-safe:

```
- public Set<Position> mice() { return mice; }
```

```

- public Set<Position> obstacles() { return obstacles; }
- public Set<Position> turbo() { return turbo; }
- public Map<Position, Position> teleports() { return teleports; }

```

- Locks granulares específicos:

```

- public MoveResult step(Snake snake) {
-     Objects.requireNonNull(snake, "snake");
-     var head = snake.head();
-     var dir = snake.direction();
-     Position next = new Position(head.x() + dir.dx, head.y() + dir.dy).wrap(width, height);
-
-     if (obstacles.contains(next)) return MoveResult.HIT_OBSTACLE;
-
-     boolean teleported = false;
-     if (teleports.containsKey(next)) {
-         next = teleports.get(next);
-         teleported = true;
-     }
-
-     boolean ateMouse = false;
-     boolean ateTurbo = turbo.remove(next);
-
-     miceLock.lock();
-     try {
-         ateMouse = mice.remove(next);
-         snake.advance(next, ateMouse);
-     } finally {
-         miceLock.unlock();
-     }
-
-     if (ateMouse) {
-         itemGenerationLock.lock();
-         try {
-             mice.add(randomEmpty());
-             obstacles.add(randomEmpty());
-             if (ThreadLocalRandom.current().nextDouble() < 0.2) {
-                 turbo.add(randomEmpty());
-             }
-         } finally {
-             itemGenerationLock.unlock();
-         }
-     }
-
-     if (ateTurbo) return MoveResult.ATE_TURBO;
-     if (ateMouse) return MoveResult.ATE_MOUSE;
-     if (teleported) return MoveResult.TELEPORTED;
-     return MoveResult.MOVED;
- }

```

- }

Sincronización Granular en `Board.step()`

Prioridad: MEDIA

Riesgo Original:

- Método *synchronized step()* bloqueaba todo el tablero.
- Una serpiente bloqueaba movimiento de todas las demás.
- Rendimiento limitado en escenarios multi-serpiente.

Solución Implementada:

- Dividido *synchronized step()* en regiones críticas específicas.
- Agregados *ReentrantLock miceLock* e *itemGenerationLock*.
- Operaciones de solo lectura sin sincronización.
- *Locks* granulares solo para operaciones que requieren atomicidad.

Por qué esta solución:

- Múltiples serpientes pueden moverse simultáneamente cuando es seguro.
- *Locks* específicos reducen *contention* y mejoran *throughput*.
- Lecturas concurrentes de obstáculos y *teleports* sin bloqueos.
- Mejor utilización de CPU en sistemas multi-core.
 - Sincronización granular específica:

```
• public MoveResult step(Snake snake) {  
•     Objects.requireNonNull(snake, "snake");  
•     var head = snake.head();  
•     var dir = snake.direction();  
•     Position next = new Position(head.x() + dir.dx, head.y() + dir.dy).wrap(width, height);  
•  
•     if (obstacles.contains(next)) return MoveResult.HIT_OBSTACLE;  
•  
•     boolean teleported = false;  
•     if (teleports.containsKey(next)) {  
•         next = teleports.get(next);  
•         teleported = true;  
•     }  
•  
•     boolean ateMouse = false;  
•     boolean ateTurbo = turbo.remove(next);  
•  
• }
```

```

• miceLock.lock();
• try {
•     ateMouse = mice.remove(next);
•     snake.advance(next, ateMouse);
• } finally {
•     miceLock.unlock();
• }
•
•
•
• if (ateMouse) {
•     itemGenerationLock.lock();
•     try {
•         mice.add(randomEmpty());
•         obstacles.add(randomEmpty());
•         if (ThreadLocalRandom.current().nextDouble() < 0.2) {
•             turbo.add(randomEmpty());
•         }
•     } finally {
•         itemGenerationLock.unlock();
•     }
• }
•
• if (ateTurbo) return MoveResult.ATE_TURBO;
• if (ateMouse) return MoveResult.ATE_MOUSE;
• if (teleported) return MoveResult.TELEPORTED;
• return MoveResult.MOVED;
• }

```

Lista Thread-Safe en SnakeApp

Prioridad: BAJA

Riesgo Original:

- *ArrayList* vulnerable a *ConcurrentModificationException*.
- Accesos concurrentes entre *UI thread* e *initialization thread*.
- *ArrayList* sin protección contra iteración concurrente.

Solución Implementada:

- *ArrayList* → *CopyOnWriteArrayList* para *snakes* y *snakeRunners*.
- Eliminada sincronización explícita en *togglePause()*.

Por qué esta solución:

- *CopyOnWriteArrayList* optimizada para lecturas frecuentes (*UI updates*).
- Thread-safe sin *locks* explícitos para mejor rendimiento en UI.
- Iteradores no lanzan *ConcurrentModificationException*.
- Escrituras atómicas durante inicialización.

- Listas thread-safe optimizadas:

```

import java.util.List;
import java.util.Map;
import java.util.concurrent.Executors;
import java.util.concurrent.CopyOnWriteArrayList;

public final class SnakeApp extends JFrame {
    private final Board board;
    private final GamePanel gamePanel;
    private final JButton actionButton;
    private final GameClock clock;
    private final java.util.List<Snake> snakes = new CopyOnWriteArrayList<>();
    private final java.util.List<SnakeRunner> snakeRunners = new CopyOnWriteArrayList<>();

    for (Snake snake : snakes) {
        SnakeRunner runner = new SnakeRunner(snake, board);
        snakeRunners.add(runner);
        clock.addListener(runner);
        exec.submit(runner);
    }

    private void togglePause() {
        if ("Action".equals(actionButton.getText())) {
            actionButton.setText("Resume");
            clock.pause();
        } else {
            actionButton.setText("Action");
            clock.resume();
        }
    }

    this.gamePanel = new GamePanel(board, () -> snakes);
}

```

Coordinación GameClock-Serpientes

Prioridad: ALTA

Riesgo Original:

- *Timing* independiente entre *GameClock* y serpientes.
- *Pause/resume* descoordinado causaba estados inconsistentes.

- Falta de sincronización entre *UI updates* y *snake movement*.

Solución Implementada:

- Agregada interfaz *GameClockListener* con métodos *onTick()*, *onPause()*, *onResume()*.
- *GameClock* notifica automáticamente a todos los *listeners* registrados.
- Sistema de registro/desregistro de *listeners* thread-safe.

Por qué esta solución:

- Coordinación centralizada elimina estados inconsistentes.
- Un solo comando controla todo el juego (*pause/resume*).
- Escalabilidad para cualquier número de serpientes.
- Separación clara entre *timing UI* y *timing game logic*.
 - Sistema de listeners coordinado:

```

• public final class GameClock implements AutoCloseable {
•     private final List<GameClockListener> listeners = new CopyOnWriteArrayList<>();
•
•     public void start() {
•         scheduler.scheduleAtFixedRate(() -> {
•             if (state.get() == GameState.RUNNING) {
•                 tick.run();
•                 notifyListeners();
•             }
•         }, 0, periodMillis, TimeUnit.MILLISECONDS);
•     }
•
•     public void pause() {
•         state.set(GameState.PAUSED);
•         listeners.forEach(GameClockListener::onPause);
•     }
•
•     public void resume() {
•         state.set(GameState.RUNNING);
•         listeners.forEach(GameClockListener::onResume);
•     }
•
•     public void addListener(GameClockListener listener) {
•         listeners.add(listener);
•     }
•
•     private void notifyListeners() {
•         listeners.forEach(GameClockListener::onTick);
•     }
•
•     @FunctionalInterface
•     public interface GameClockListener {

```

```

• void onTick();
• default void onPause() {}
• default void onResume() {}
• default void onStop() {}
• }
• }

```

▪ Listener coordinado:

```

• public final class SnakeRunner implements Runnable, GameClock.GameClockListener {
•     private volatile boolean clockTick = false;
•
•     @Override
•     public void run() {
•         while (!stopped) {
•             synchronized (pauseLock) {
•                 while (!clockTick && !stopped) {
•                     int sleepTime = (turboTicks > 0) ? turboSleepMs : baseSleepMs;
•                     pauseLock.wait(sleepTime);
•                 }
•                 clockTick = false;
•             }
•         }
•     }
•
•     @Override
•     public void onTick() {
•         synchronized (pauseLock) {
•             clockTick = true;
•             pauseLock.notify();
•         }
•     }
•
•     @Override
•     public void onPause() {
•         pause();
•     }
•
•     @Override
•     public void onResume() {
•         resume();
•     }
• }

```


- Registro automático:

```
• var exec = Executors.newVirtualThreadPerTaskExecutor();
• for (Snake snake : snakes) {
•     SnakeRunner runner = new SnakeRunner(snake, board);
•     snakeRunners.add(runner);
•     clock.addListener(runner);
•     exec.submit(runner);
• }
•
• private void togglePause() {
•     if ("Action".equals(actionButton.getText())) {
•         actionButton.setText("Resume");
•         clock.pause();
•     } else {
•         actionButton.setText("Action");
•         clock.resume();
•     }
• }
• }
```

3. Control de ejecución seguro (UI):

- Implementa la UI con Iniciar / Pausar / Reanudar (ya existe el botón *Action* y el reloj *GameClock*).

Para ejecutar el proyecto se utilizan los siguientes comandos:

```
mvn clean verify
mvn -q -DskipTests exec:java -Dsnakes=2
```

```
[INFO] -----
PS C:\Users\David\Documents\Lab2_SnakeRace-Java21_ARSW_GROUP> mvn clean verify
>> mvn -q -DskipTests exec:java -Dsnakes=2
```

Lo primero que se aprecia en la interfaz es un campo totalmente estático, con el tiempo de la partida en 0 ubicado en la parte superior derecha. En la zona central inferior se encuentran dos botones: uno para iniciar el juego y otro para finalizarlo.



Al presionar el botón Iniciar, este cambia su etiqueta a Pausar, mientras que el botón de Salir permanece igual. En ese momento, el tiempo de la partida comienza a correr.

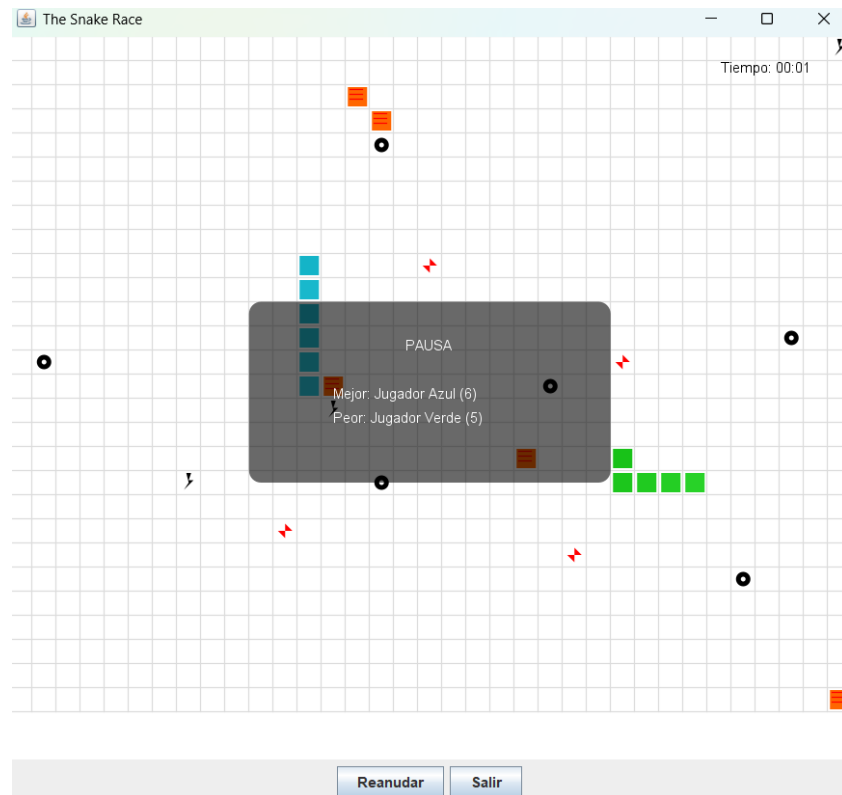


Si se selecciona la opción Pausar, el mismo botón cambia nuevamente a Reanudar.



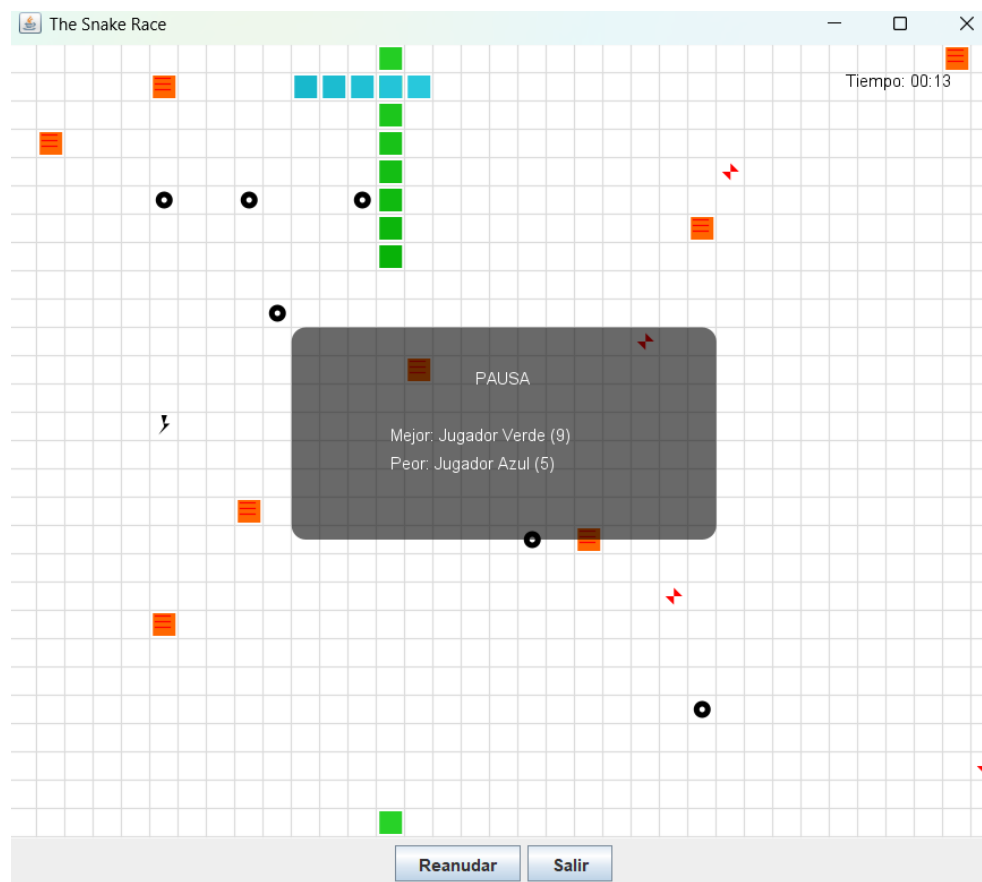
- Al Pausar, muestra de forma consistente (sin *tearing*):
 - La serpiente viva más larga.
 - La peor serpiente (La que murió primero).

Lo que implementamos fue un ranking de las serpientes según sus colores (en este caso, azul y verde). Al pausar el juego, se mostrará un mensaje en pantalla indicando la posición de cada una: si la serpiente azul es la mejor, se mostrará su puntuación o longitud, y lo mismo ocurrirá con la serpiente con menor rendimiento. Además, al pausar, el tiempo de la partida se detiene y la imagen permanece fija de manera estable, sin presentar *tearing*.



- Considera que la suspensión no es instantánea; coordina para que el estado mostrado no quede “a medias”.

La suspensión del juego no ocurre de manera instantánea; no obstante, esto se resolvió mediante la implementación del ranking de las serpientes junto con el reloj de la partida, que coordina la actualización del estado antes de mostrarlo en pantalla. De esta forma, al pausar el juego se detiene el tiempo, se actualiza la información del ranking y se garantiza que la interfaz no quede “a medias”, mostrando datos consistentes y estables.



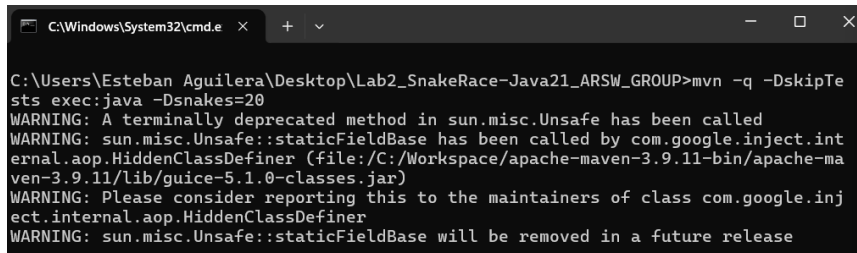
4. Robustez bajo carga:

Ejecuta con N alto (-Dsnakes=20 o más) y/o aumenta la velocidad.

El juego no debe romperse: sin `ConcurrentModificationException`, sin lecturas inconsistentes, sin *deadlocks*.

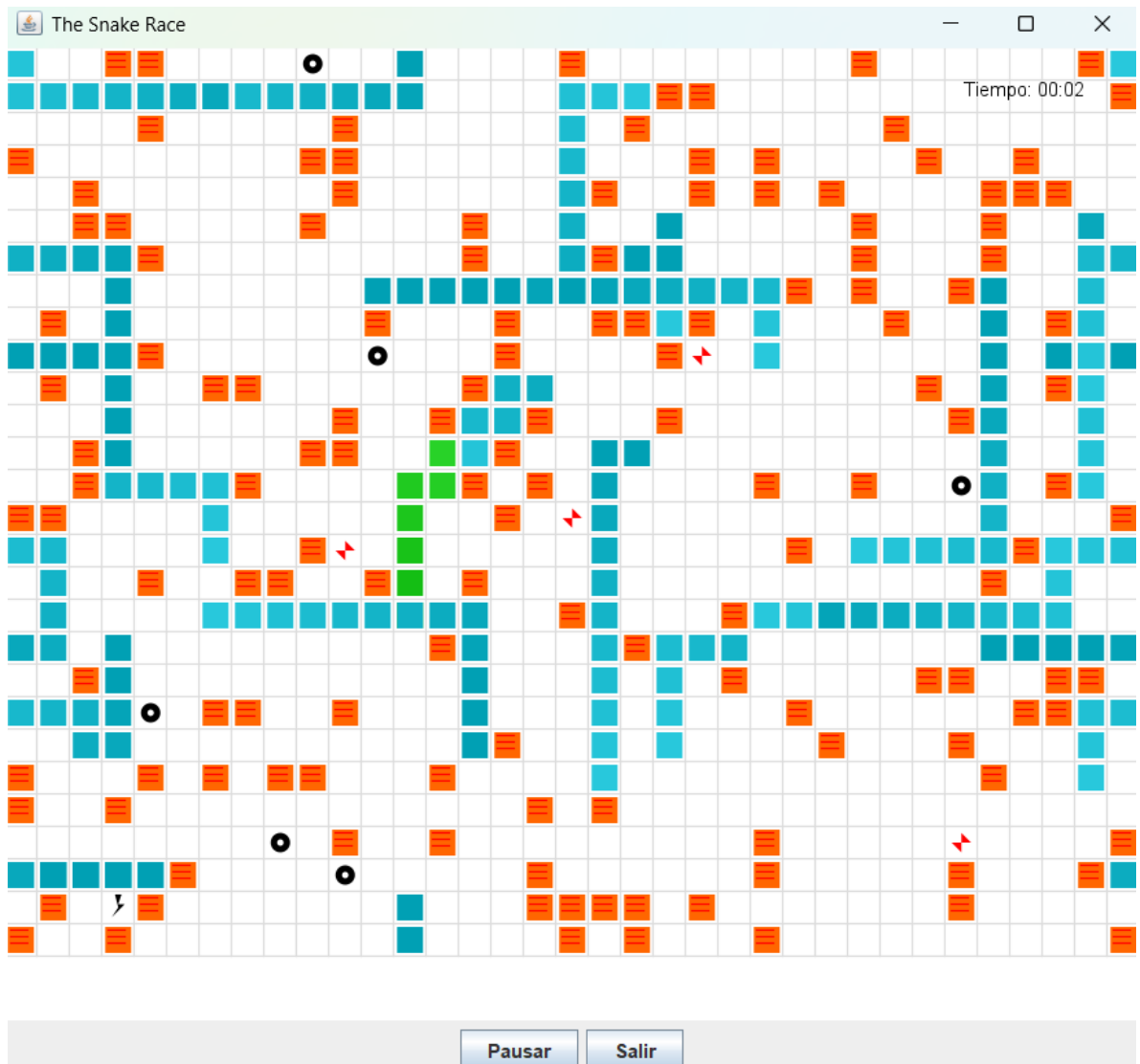
Si habilitas teleports y turbo, verifica que las reglas no introduzcan carreras.

- Se ejecutó el juego con 20 serpientes mediante el comando: `mvn -q -DskipTests exec:java -Dsnakes=20`



```
C:\Windows\System32\cmd.e  X + v
C:\Users\Esteban Aguilera\Desktop\Lab2_SnakeRace-Java21_ARSW_GROUP>mvn -q -DskipTests exec:java -Dsnakes=20
WARNING: A terminally deprecated method in sun.misc.Unsafe has been called
WARNING: sun.misc.Unsafe::staticFieldBase has been called by com.google.inject.internal.aop.HiddenClassDefiner (file:/C:/Workspace/apache-maven-3.9.11-bin/apache-maven-3.9.11/lib/guice-5.1.0-classes.jar)
WARNING: Please consider reporting this to the maintainers of class com.google.inject.internal.aop.HiddenClassDefiner
WARNING: sun.misc.Unsafe::staticFieldBase will be removed in a future release
```

- Durante la prueba no se presentaron errores de concurrencia:
 - Sin `ConcurrentModificationException` , gracias al uso de colecciones seguras como `CopyOnWriteArrayList`, que permite iterar y modificar sin lanzar excepciones
 - Sin lecturas inconsistentes , ya que las variables compartidas como es el caso de `stopped`, `paused` y `direction` están declaradas como `volatile`, esto garantiza la visibilidad entre hilos
 - Sin deadlocks, porque en cada `SnakeRunner` solo se utiliza un único lock (`pauseLock`), evitando interbloqueos
- Las funciones `teleport` y `turbo` no generaron carreras de datos ya que la lógica de cada serpiente esta encapsulada en su propio hilo
- El resultado del juego se mantiene estable y consistente incluso si se aumenta mas la cantidad de serpientes, como se evidencia en la imagen



CONCLUSIONES

La sincronización con wait/notify y el uso del GameClock eliminaron las esperas activas, redujeron el consumo de CPU y permitieron movimientos coordinados de las serpientes. El empleo de colecciones concurrentes como ConcurrentHashMap y CopyOnWriteArrayList evitó excepciones y mejoró la escalabilidad. Asimismo, las variables volatile garantizaron visibilidad entre hilos y evitaron resultados inesperados en las pausas y reanudaciones.

Las pruebas con 20 serpientes demostraron robustez: no se presentaron deadlocks, carreras de datos ni inconsistencias, incluso con una cantidad de 20 serpientes. El uso de ReentrantLock en regiones críticas evitó bloqueos globales y permitió que varias serpientes se movieran simultáneamente de forma eficiente.

El laboratorio mostró que el éxito en el manejo de concurrencia no depende únicamente de crear más hilos, sino de diseñar arquitecturas funcionales, escalables y con sincronización.

En conclusión, la experiencia adquirida no solo fortaleció la comprensión de la programación concurrente en Java, sino que también resaltó la importancia de la planificación cuidadosa y el diseño como pilares fundamentales para el desarrollo de sistemas confiables y eficientes.

WEBGRAFIA

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). Java concurrency in practice. Addison-Wesley.