

**Tema**

HILOS CON JAVA

CONCURRENCIA Y PARALELISMO

**Tutor**

Ing. Eduardo Mauricio Campaña Ortega

MIS. MDU.CCNA. CCIA.

PhD. (c) Ingeniería de Software

PhD. (c) Seguridad Información

**Fecha**

27/11/2022

HILOS CON JAVA: CONCURRENCIA Y SINCRONIZACIÓN

[1. INTRODUCCIÓN 4](#_Toc120696300)

[2. OBJETIVOS 4](#_Toc120696301)

[2.1 OBJETIVO GENERAL 4](#_Toc120696302)

[2.2 OBJETIVOS ESPECÍFICOS 4](#_Toc120696303)

[3. MARCO TEÓRICO 4](#_Toc120696304)

[3.1 HILOS 4](#_Toc120696305)

[3.1.1 HILOS CON JAVA 5](#_Toc120696306)

[3.1.2 CONCURRENCIA Y PARALELISMO 5](#_Toc120696307)

[3.2 PATRÓN MODELO VISTA CONTROLADOR 6](#_Toc120696308)

[3.2.1 MODELO 6](#_Toc120696309)

[3.2.2 VISTA 6](#_Toc120696310)

[3.2.3 CONTROLADOR 7](#_Toc120696311)

[4. PARTE PRÁCTICA 7](#_Toc120696312)

[4.1 CODIFICACIÓN VIDEO 168 7](#_Toc120696313)

[4.2 CODIFICACIÓN VIDEO 169 10](#_Toc120696314)

[4.3 CODIFICACIÓN VIDEO 170 12](#_Toc120696315)

[4.4 CODIFICACIÓN FINAL DE LA PRACTICA 13](#_Toc120696316)

[4.5 FUNCIONALIDAD 22](#_Toc120696317)

[4.5.1 ESTRUCTURA DE LA APLICACIÓN 22](#_Toc120696318)

[4.5.2 EJECUCIÓN DE LA APLICACIÓN 22](#_Toc120696319)

[5 CONCLUSIONES 25](#_Toc120696320)

[6 RECOMENDACIONES 25](#_Toc120696321)

[7 REFERENCIAS 25](#_Toc120696322)

**INDICE DE IMÁGENES**

[Figura 1. Hilos y Procesos 5](#_Toc120664124)

[Figura 2. Logo identificativo de Java 5](#_Toc120664125)

[Figura 3. Concurrencia 6](#_Toc120664126)

[Figura 4. Paralelismo 6](#_Toc120664127)

[Figura 5. Modelo Vista Controlador 7](#_Toc120664128)

[Figura 6. Codificación clase PelotaVista 8](#_Toc120664129)

[Figura 7. Codificación clase PelotaPanelVista 9](#_Toc120664130)

[Figura 8. Codificación clase PelotaModelo 9](#_Toc120664131)

[Figura 9. Codificación clase PelotaControlador 10](#_Toc120664132)

[Figura 10. Codificación clase HiloPelotaControlador 10](#_Toc120664133)

[Figura 11. Codificación clase AppMain 11](#_Toc120664134)

[Figura 12. Codificación clase PelotaVista 12](#_Toc120664135)

[Figura 13. Codificación método detener 12](#_Toc120664136)

[Figura 14. Codificación método run 12](#_Toc120664137)

[Figura 15. Codificación clase PelotaVista 13](#_Toc120664138)

[Figura 16. Codificación de los botones para controlar pelotas 13](#_Toc120664139)

[Figura 17. Codificación método comenzarJuego 13](#_Toc120664140)

[Figura 18. Codificación método detener 14](#_Toc120664141)

[Figura 19. Estructura de la aplicación 23](#_Toc120664142)

[Figura 20. Ventana inicial sin pelotas 23](#_Toc120664143)

[Figura 21. Ventana con pelotas gestionadas por hilos 24](#_Toc120664144)

[Figura 22. Ventana inicial sin pelotas con la opción detener 24](#_Toc120664145)

[Figura 23. Ventana con pelotas y sus hilos detenidos 25](#_Toc120664146)

[Figura 24. Ventana inicial sin las 3 pelotas 25](#_Toc120664147)

[Figura 25. Ventana con las 3 pelotas gestionadas por hilos 25](#_Toc120664148)

**ÍNDICE DE TABLAS**

[Tabla 1. Codificación del archivo Main.java 15](file:///C:\Users\santi\Downloads\Informes\videospelotas.docx#_Toc120664883)

[Tabla 2. Codificación del archivo PelotaModelo.java 15](file:///C:\Users\santi\Downloads\Informes\videospelotas.docx#_Toc120664884)

[Tabla 3. Codificación del archivo PelotaPanelVista.java 16](file:///C:\Users\santi\Downloads\Informes\videospelotas.docx#_Toc120664885)

[Tabla 4. Codificación del archivo PelotaVista.java 17](file:///C:\Users\santi\Downloads\Informes\videospelotas.docx#_Toc120664886)

[Tabla 5. Codificación del archivo HiloPelotaControlador.java 20](file:///C:\Users\santi\Downloads\Informes\videospelotas.docx#_Toc120664887)

[Tabla 6. Codificación del archivo PelotaControlador.java 21](file:///C:\Users\santi\Downloads\Informes\videospelotas.docx#_Toc120664888)

# INTRODUCCIÓN

Las aplicaciones de software siempre necesitan realizar varias tareas a la vez y no una después de que finalice la otra, esto puede provocar que el sistema y el usuario no sean eficientes al interactuar el uno con el otro. En un flujo de tareas normal una se ejecutará después de la otra, pero lo ideal es mantener varios flujos de ejecución para poder realizar una tarea sin necesidad de esperar a las otras (Nakayama, 2009).

Una aplicación comúnmente debe realizar varias operaciones a la vez como descargar información o contenido, actualizar información del catálogo de productos, sincronizar información con otros usuarios, etc. Los hilos permiten ejecutar varios procesos de forma paralela, tomando en cuenta el acceso a los recursos para que no se provoque una pérdida de información.

# OBJETIVOS

## 2.1 OBJETIVO GENERAL

Analizar y comprender la definición, usos y funcionamiento de los hilos, concurrencia y paralelismo utilizando el lenguaje de programación JAVA y el IDE Apache NetBeans.

## 2.2 OBJETIVOS ESPECÍFICOS

* Realizar un estudio del estado del arte sobre los hilos, concurrencia y paralelismo en el lenguaje de programación JAVA.
* Analizar la importancia de los hilos en aplicaciones y programas de software.
* Desarrollar un ejemplo práctico para comprender el uso de los hilos y su comportamiento con una aplicación JAVA.

# MARCO TEÓRICO

## HILOS

Un hilo es un solo flujo de ejecución de instrucciones dentro de un proceso, debido a que no pueden ejecutarse por sí solos, requieren la supervisión de un proceso (Nakayama, 2009). Existen varios lenguajes de programación que permiten crear y gestionar hilos, entre los más conocidos están JAVA y C++.

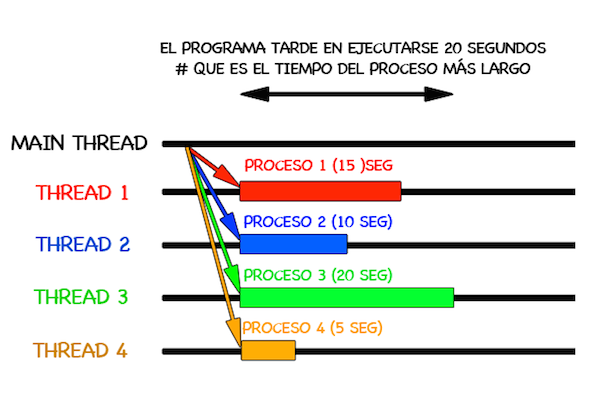
**

Figura 1. Hilos y Procesos

### **HILOS CON JAVA**

JAVA es un lenguaje de programación de altor nivel con tipado estático de datos, está en casi todos los sistemas y dispositivos informáticos convirtiéndolo en uno de los lenguajes más usados y demandados de la actualidad. Soporta múltiples paradigmas de programación y ha sido la base para crear lenguajes de programación modernos como SCALA o Kotlin.

El lenguaje de programación JAVA permite crear hilos y administrarlos gracias a su JVM (Java Virtual Machine). Java es capaz de manejar múltiples hilos con varios flujos de instrucciones y un flujo de datos de forma simultánea (Díaz, 2022).



Figura 2. Logo identificativo de Java

### **CONCURRENCIA Y PARALELISMO**

La concurrencia es la capacidad del CPU para ejecutar varias actividades o procesos al mismo tiempo (Blancarte, 2017). En la concurrencia los procesos no están relacionado, por lo tanto, cualquiera de los procesos puede iniciar y terminar sin afectar a los demás.

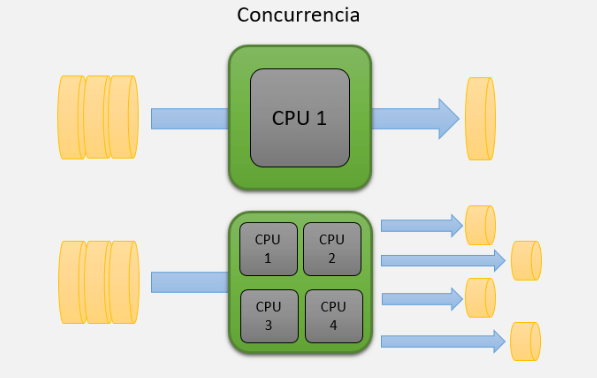


Figura 3. Concurrencia

El paralelismo consiste en tomar el problema inicial, dividirlo en fracciones más pequeñas y procesarlas aprovechando al máximo la capacidad el procesador (Blancarte, 2017).la principal diferencia con la concurrencia es que todos los procesos están relacionados para resolver el mismo problema.

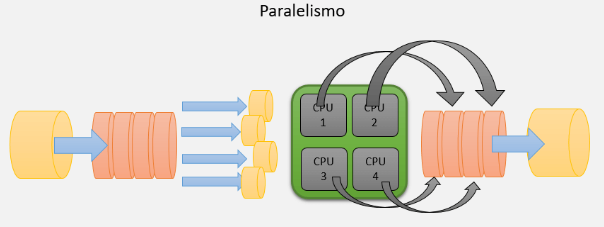


Figura 4. Paralelismo

## PATRÓN MODELO VISTA CONTROLADOR

Es un patrón de arquitectura para el desarrollo de aplicaciones software donde su objetivo es el separar la lógica del negocio de la interfaz de usuario lo cual lo hace escalable facilitando la evolución por separado de sus partes, incrementa la reutilización y flexibilidad.

Tuvo su origen en 1979 y es también conocido como patrón MVC, divide una aplicación software en tres grandes partes bien diferenciadas las cuales son Modelo, Vista y Controlador.

### **MODELO**

Es una capa donde se localiza la funcionalidad central y los datos, se comunica con el controlador y la base de datos.

Es la capa donde se trabaja con los datos, por tanto, contendrá mecanismos para acceder a la información y también para actualizar su estado. Los datos los tendremos habitualmente en una base de datos, por lo que en los modelos tendremos todas las funciones que accederán a las tablas y harán los correspondientes selects, updates, inserts, etc.

No obstante, cabe mencionar que cuando se trabaja con MCV lo habitual también es utilizar otras librerías como PDO o algún ORM como Doctrine, que nos permiten trabajar con abstracción de bases de datos y persistencia en objetos. Por ello, en vez de usar directamente sentencias SQL, que suelen depender del motor de base de datos con el que se esté trabajando, se utiliza un dialecto de acceso a datos basado en clases y objetos.

### **VISTA**

Es la capa donde se muestra la información al usuario donde es posible definir una o más vista de usuario en un software, se comunica con el controlador y el usuario.

Las vistas, como su nombre nos hacen entender, contienen el código de nuestra aplicación que va a producir la visualización de las interfaces de usuario, o sea, el código que nos permitirá renderizar los estados de nuestra aplicación en HTML. En las vistas nada más tenemos los códigos HTML y PHP que nos permite mostrar la salida.

En la vista generalmente trabajamos con los datos, sin embargo, no se realiza un acceso directo a éstos. Las vistas requerirán los datos a los modelos y ellas se generarán la salida, tal como nuestra aplicación requiera.

### **CONTROLADOR**

Es la capa donde se manejan las entradas del usuario se separa la representación interna de la información y la forma en la que se le muestra al usuario, se comunica con el modelo.

Contiene el código necesario para responder a las acciones que se solicitan en la aplicación, como visualizar un elemento, realizar una compra, una búsqueda de información, etc.

En realidad, es una capa que sirve de enlace entre las vistas y los modelos, respondiendo a los mecanismos que puedan requerirse para implementar las necesidades de nuestra aplicación. Sin embargo, su responsabilidad no es manipular directamente datos, ni mostrar ningún tipo de salida, sino servir de enlace entre los modelos y las vistas para implementar las diversas necesidades del desarrollo.

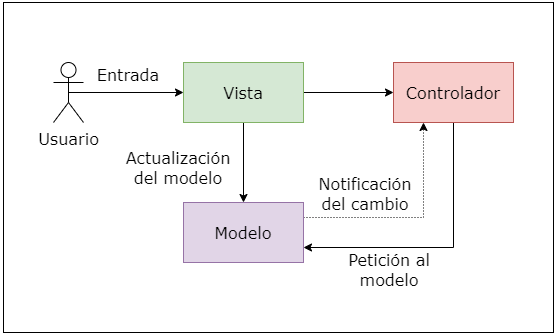


Figura 5. Modelo Vista Controlador

# PARTE PRÁCTICA

## CODIFICACIÓN VIDEO 168

**Programación de un hilo**

Esta práctica consiste en simular el movimiento de una pelota que rebota por los bordes del frame/lámina y se mantiene rebotando hasta que se cierre la aplicación, además, existen 2 botones con funcionalidades propias para cada uno, uno que permite generar una nueva pelota y otro que permite terminar la ejecución de la aplicación.

**Paquete Vista**

A continuación, se detallará el contenido del paquete vista, donde se podrá encontrar 2 archivos/vistas de las cuales, la primera se encargará de mostrar los botones de control y la lámina donde se alojarán estos, la segunda vista está encargada de mostrar la lámina por la cual se mueven las pelotas generadas.

El contenido de la primera vista “PelotaVista” se puede observar en la siguiente figura, donde se logra apreciar que se extiende de la función JFrame y se hace uso de componentes propios de la función como el JPanel y los JButton.

Texto

Descripción generada automáticamente

Figura 6. Codificación clase PelotaVista

Dentro del contenido de la segunda vista “PelotaPanelVista” se pueden encontrar elementos similares a la vista detallada anteriormente, ya que utiliza componentes y se extiende de JPanel, posteriormente se declara un arreglo de pelotas que se mostrarán al momento de la ejecución del aplicativo, además se genera la lámina con restricciones para que la pelota generada no se salga de los límites establecidos.

Texto

Descripción generada automáticamente

Figura 7. Codificación clase PelotaPanelVista

**Paquete Modelo**

La práctica fue desarrollada usando un solo modelo llamado “PelotaModelo” cuyo contenido será descrito a continuación:

Dentro de la clase PelotaModelo encontramos la declaración de atributos correspondientes a las coordenadas que servirá para el movimiento de las pelotas, de igual manera se declaran atributos que sirven para invertir el movimiento de la pelota cada vez que la pelota choque con una pared de la lámina.

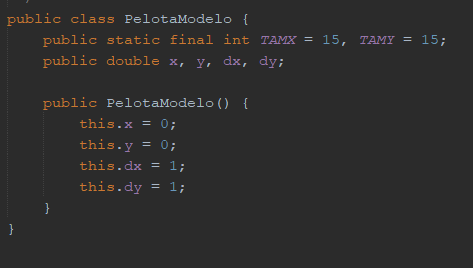


Figura 8. Codificación clase PelotaModelo

**Paquete Controlador**

Dentro del paquete controlador, se encuentra la clase “PelotaControlador” dentro de la cual podemos encontrar el método moverPelota que recibe 2 parámetros, uno para los límites de la lámina a crear y uno de tipo PelotaModelo para la creación de las pelotas en la ejecución, dentro de este método creado podemos encontrar la función encargada del movimiento de la pelota, de igual manera se encarga del movimiento de la pelota al momento de colisionar con una pared de la lámina invirtiendo su posición y orientación.

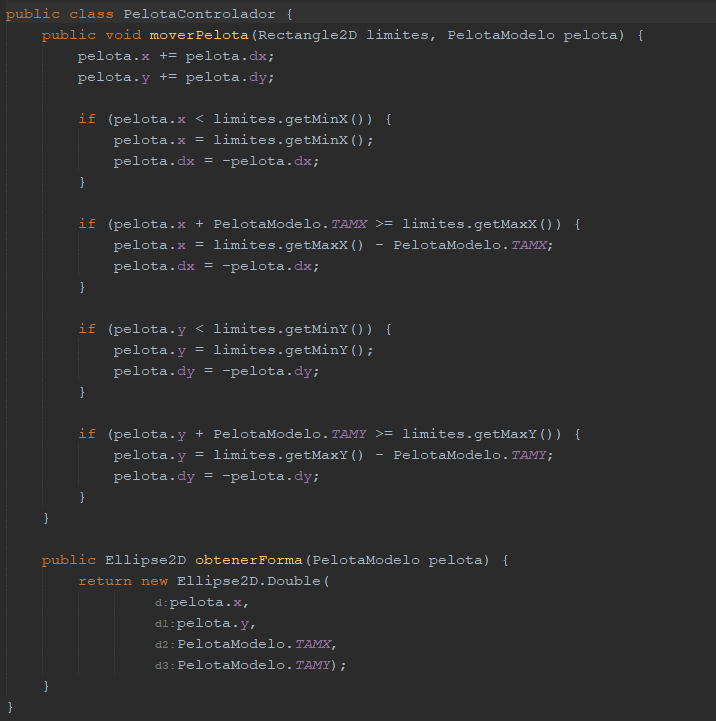


Figura 9. Codificación clase PelotaControlador

Como segundo controlador encontramos al denominado “HIloPelotaControlador” donde encontramos declaraciones de un modelo denominado “pelota” y un controlador denominado “pelotaController”, además de un componente extra.

Encontramos un método constructor para asignar valores a cada uno de los datos declarados, donde además encontraremos un método de tipo run() que contiene un ciclo for para simular el movimiento de la pelota, además utiliza dentro del ciclo al método moverPelota y a Paint para obtener los límites y dibujar la lámina respectivamente.

También se encuentra el método sleep() dentro de esta función, el cual se encarga de pausa la ejecución por milisegundos, lo que nos permitirá apreciar el movimiento de las pelotas.

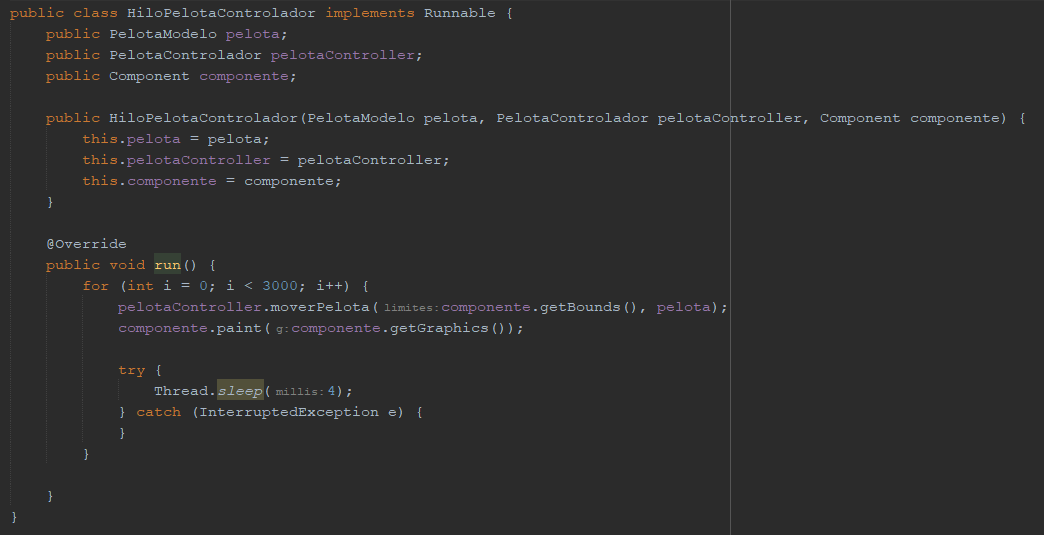


Figura 10. Codificación clase HiloPelotaControlador

**Main**

Finalmente, nos encontramos con el método main el cual se encarga de inicializar todos los datos necesarios para la ejecución de la aplicación, en este encontramos una declaración de un dato de tipo JFrame encargado de generar las láminas donde se podrá observar la ejecución, un método que sirve para cerrar el cuadro de ejecución cuando se presione la X de la ventana y un método para hacer visibles las láminas.

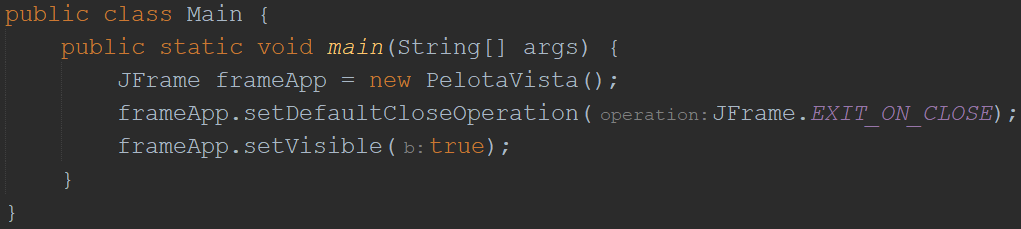


Figura 11. Codificación clase AppMain

## CODIFICACIÓN VIDEO 169

La práctica del video 169 consiste en desarrollar una aplicación para observar la ejecución de un hilo, pero utilizando métodos propios de la clase “Thread”, como, por ejemplo:

* interrupt: Sirve para solicitar la interrupción de un hilo que está en ejecución.
* interrupted: Sirve para solicitar el estado de un hilo, dándonos como resultado un valor de tipo booleano.
* isInterrupted: Similar al método interrupted, con la diferencia que este es declarado de tipo static.
* stop: Método obsoleto que servia para solicitar la detención de un hilo.

Para el desarrollo de esta práctica es necesario proseguir con la practica anterior, aumentando ciertos aspectos y funcionalidades.

**Paquete Vista**

Dentro del paquete vista, en el método constructor de la clase “PelotaVista” se debe agregar un nuevo botón que nos permitirá detener la ejecución de un hilo, este invocara al método detener(), declarado dentro de la misma clase solicitando la detención del hilo con el método interrupt().

Text

Description automatically generated

Figura 12. Codificación clase PelotaVista

Texto

Descripción generada automáticamente con confianza media

Figura 13. Codificación método detener

**Paquete Modelo**

En este paquete se mantendrá la estructura y los mismos datos, no se debe cambiar nada en esta sección.

**Paquete controlador**

Dentro de HiloPelotaControlador se cambia el uso de ThreadSleep y ahora se utiliza el ciclo while junto a la función isInterrupted().

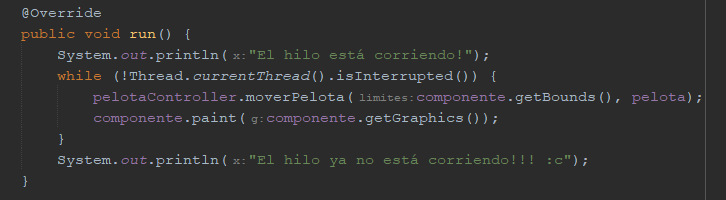


Figura 14. Codificación método run

**Main**

En esta sección se debe mantener la misma estructura, no se debe cambiar nada con respecto a la práctica anterior.

## CODIFICACIÓN VIDEO 170

Para la realización de esta práctica se trabaja sobre los proyectos desarrollados en prácticas anteriores, cambiando aspectos muy puntuales como se describirá a continuación:

**Paquete Vista**

Dentro de la vista denominada “PelotaVista” se agregarán 3 hilos y 3 botones para generar el mismo número de pelotas.

Text

Description automatically generated

Figura 15. Codificación clase PelotaVista

Como se agregaron botones a la lámina es necesario inicializarlos y crearlos como se muestra a continuación:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Figura 16. Codificación de los botones para controlar pelotas

De igual manera, es necesario realizar cambios a la función comenzarJuego() para que permita recibir un parámetro y especificar el número de hilo que deberá ser creado y ejecutado.

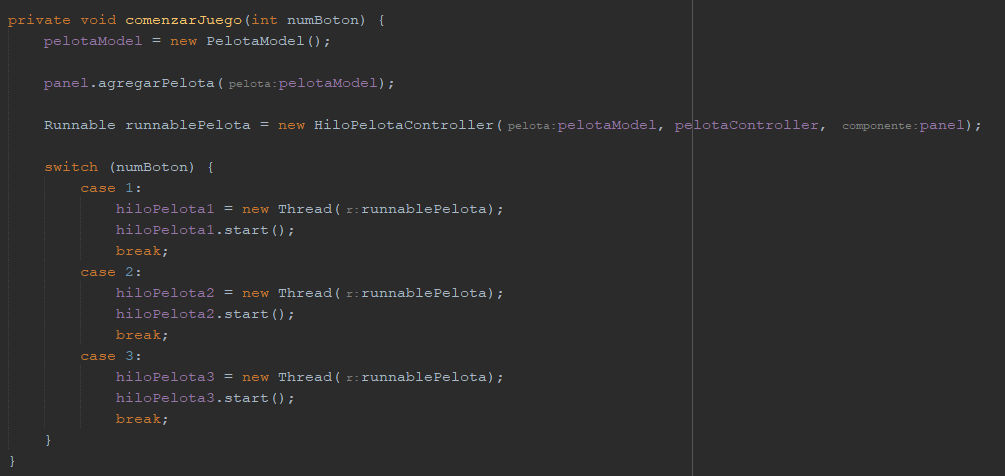


Figura 17. Codificación método comenzarJuego

Conjuntamente se necesita cambiar la función detener() para que permita recibir un parámetro y permita especificar el hilo que se va a interrumpir como se muestra a continuación:

Texto

Descripción generada automáticamente

Figura 18. Codificación método detener

**Paquete Modelo**

En este paquete se mantendrá todo tal cual las practicas anteriores.

**Paquete Controlador**

En este paquete se mantendrá todo tal cual las practicas anteriores.

**Main**

Dentro de la función principal no se tiene cambios, por lo que se mantiene la estructura ya realizada.

## CODIFICACIÓN FINAL DE LA PRACTICA

package ec.edu.monster;

import javax.swing.JFrame;

import ec.edu.monster.vista.PelotaVista;

/\*\*

\* @author Grupo5

\*/

public class Main {

public static void main(String[] args) {

JFrame frameApp = new PelotaVista();

frameApp.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frameApp.setVisible(true);

}

}

Tabla 1. Codificación del archivo Main.java

Tabla 2. Codificación del archivo PelotaModelo.java

package ec.edu.monster.modelo;

/\*\*

\* @author Grupo5

\*/

public class PelotaModelo {

public static final int TAMX = 15, TAMY = 15;

public double x, y, dx, dy;

public PelotaModelo() {

this.x = 0;

this.y = 0;

this.dx = 1;

this.dy = 1;

}

}

Tabla 3. Codificación del archivo PelotaPanelVista.java

package ec.edu.monster.vista;

/\*\*

\*

\* @author Grupo5

\*/

import java.awt.Graphics;

import java.awt.Graphics2D;

import java.util.ArrayList;

import javax.swing.JPanel;

import ec.edu.monster.controlador.PelotaControlador;

import ec.edu.monster.modelo.PelotaModelo;

class PelotaPanelVista extends JPanel {

private final ArrayList<PelotaModelo> arrPelotas = new ArrayList<>();

private final PelotaControlador pelotaController = new PelotaControlador();

public void agregarPelota(PelotaModelo pelota) {

arrPelotas.add(pelota);

}

public void paint(Graphics g) {

super.paintComponent(g);

Graphics2D g2 = (Graphics2D) g;

arrPelotas.forEach(p -> g2.fill(pelotaController.obtenerForma(p)));

}

}

Tabla 4. Codificación del archivo PelotaVista.java

package ec.edu.monster.vista;

import ec.edu.monster.controlador.HiloPelotaControlador;

import ec.edu.monster.controlador.PelotaControlador;

import java.awt.BorderLayout;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

import ec.edu.monster.modelo.PelotaModelo;

/\*\*

\*

\* @author Grupo5

\*/

public class PelotaVista extends JFrame {

public PelotaModelo pelotaModel;

private final PelotaPanelVista panel;

public PelotaControlador pelotaController;

private Thread hiloPelota1, hiloPelota2, hiloPelota3;

private JButton btnDetenerPelota1, btnDetenerPelota2, btnDetenerPelota3;

private JButton btnComenzarPelota1, btnComenzarPelota2, btnComenzarPelota3;

public PelotaVista() {

JPanel panelBotones = new JPanel();

panel = new PelotaPanelVista();

pelotaController = new PelotaControlador();

setBounds(600, 300, 800, 350);

setTitle("Rebotes");

btnComenzarPelota1 = obtenerBoton("Hilo 1", event -> comenzarJuego(1));

btnComenzarPelota2 = obtenerBoton("Hilo 2", event -> comenzarJuego(2));

btnComenzarPelota3 = obtenerBoton("Hilo 3", event -> comenzarJuego(3));

btnDetenerPelota1 = obtenerBoton("Detener 1", event -> detener(1));

btnDetenerPelota2 = obtenerBoton("Detener 2", event -> detener(2));

btnDetenerPelota3 = obtenerBoton("Detener 3", event -> detener(3));

panelBotones.add(btnComenzarPelota1);

panelBotones.add(btnComenzarPelota2);

panelBotones.add(btnComenzarPelota3);

panelBotones.add(btnDetenerPelota1);

panelBotones.add(btnDetenerPelota2);

panelBotones.add(btnDetenerPelota3);

add(panel, BorderLayout.CENTER);

add(panelBotones, BorderLayout.SOUTH);

}

private JButton obtenerBoton(String titulo, ActionListener listener) {

JButton boton = new JButton(titulo);

boton.addActionListener(listener);

return boton;

}

private void comenzarJuego(int numBoton) {

pelotaModel = new PelotaModelo();

panel.agregarPelota(pelotaModel);

Runnable runnablePelota = new HiloPelotaControlador(pelotaModel, pelotaController, panel);

switch (numBoton) {

case 1:

hiloPelota1 = new Thread(runnablePelota);

hiloPelota1.start();

break;

case 2:

hiloPelota2 = new Thread(runnablePelota);

hiloPelota2.start();

break;

case 3:

hiloPelota3 = new Thread(runnablePelota);

hiloPelota3.start();

break;

}

}

private void detener(int numBoton) {

switch (numBoton) {

case 1:

hiloPelota1.interrupt();

break;

case 2:

hiloPelota2.interrupt();

break;

case 3:

hiloPelota3.interrupt();

break;

}

}

}

package ec.edu.monster.controlador;

import java.awt.Component;

import ec.edu.monster.modelo.PelotaModelo;

public class HiloPelotaControlador implements Runnable {

public PelotaModelo pelota;

public PelotaControlador pelotaController;

public Component componente;

public HiloPelotaControlador(PelotaModelo pelota, PelotaControlador pelotaController, Component componente) {

this.pelota = pelota;

this.pelotaController = pelotaController;

this.componente = componente;

}

@Override

public void run() {

System.out.println("El hilo está corriendo!");

while (!Thread.currentThread().isInterrupted()) {

pelotaController.moverPelota(componente.getBounds(), pelota);

componente.paint(componente.getGraphics());

try {

Thread.sleep(4);

} catch (Exception e) {

Thread.currentThread().interrupt();

}

}

System.out.println("El hilo ya no está corriendo!!! :c");

}

}

Tabla 5. Codificación del archivo HiloPelotaControlador.java

package ec.edu.monster.controlador;

import java.awt.geom.Ellipse2D;

import java.awt.geom.Rectangle2D;

import ec.edu.monster.modelo.PelotaModelo;

/\*\*

\*

\* @author Grupo5

\*/

public class PelotaControlador {

public void moverPelota(Rectangle2D limites, PelotaModelo pelota) {

pelota.x += pelota.dx;

pelota.y += pelota.dy;

if (pelota.x < limites.getMinX()) {

pelota.x = limites.getMinX();

pelota.dx = -pelota.dx;

}

if (pelota.x + PelotaModelo.TAMX >= limites.getMaxX()) {

pelota.x = limites.getMaxX() - PelotaModelo.TAMX;

pelota.dx = -pelota.dx;

}

if (pelota.y < limites.getMinY()) {

pelota.y = limites.getMinY();

pelota.dy = -pelota.dy;

}

if (pelota.y + PelotaModelo.TAMY >= limites.getMaxY()) {

pelota.y = limites.getMaxY() - PelotaModelo.TAMY;

pelota.dy = -pelota.dy;

}

}

Tabla 6. Codificación del archivo PelotaControlador.java

public Ellipse2D obtenerForma(PelotaModelo pelota) {

return new Ellipse2D.Double(

pelota.x,

pelota.y,

PelotaModelo.TAMX,

PelotaModelo.TAMY);

}

}

## FUNCIONALIDAD

### **ESTRUCTURA DE LA APLICACIÓN**

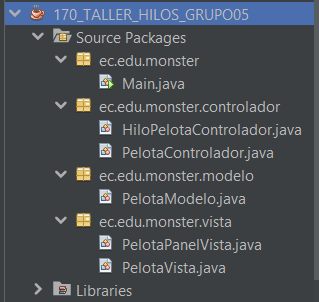


Figura 19. Estructura de la aplicación

### **EJECUCIÓN DE LA APLICACIÓN**

**Ejecución Video 168**

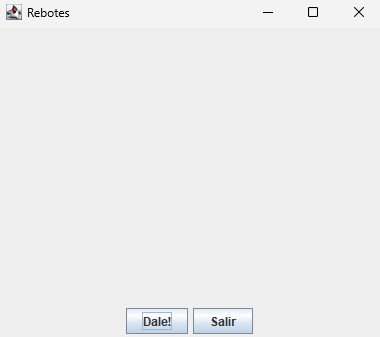


Figura 20. Ventana inicial sin pelotas

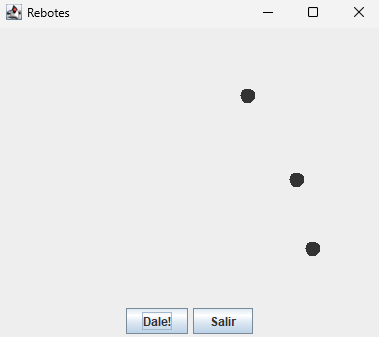


Figura 21. Ventana con pelotas gestionadas por hilos

**Ejecución Video 169**

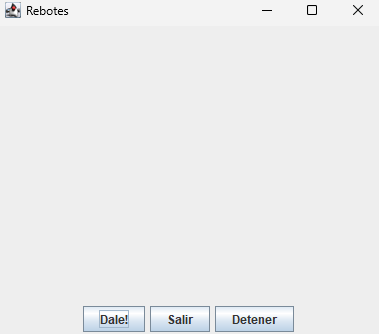


Figura 22. Ventana inicial sin pelotas con la opción detener

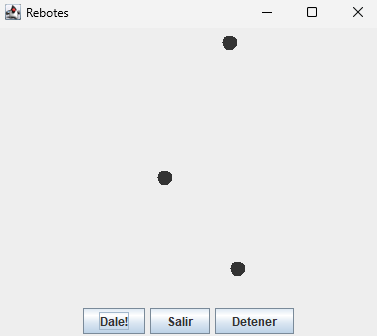


Figura 23. Ventana con pelotas y sus hilos detenidos

**Ejecución Video 170**



Figura 24. Ventana inicial sin las 3 pelotas

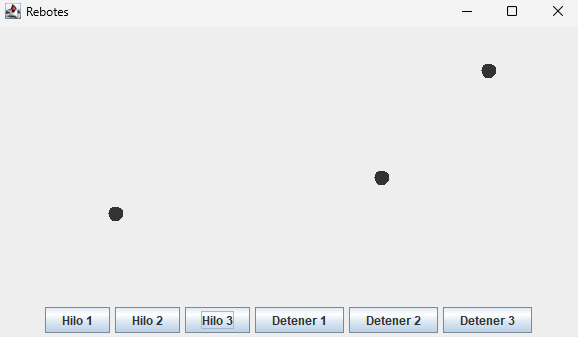


Figura 25. Ventana con las 3 pelotas gestionadas por hilos

# CONCLUSIONES

* La concurrencia y paralelismo son dos técnicas de programación, la adopción de una u otra ya depende de la naturaleza del problema a resolver, el paralelismo comúnmente se usa en supercomputadoras para resolver problemas de simulación o grandes procesamientos de datos.
* Los inicios de la programación comienzan con prácticas en el que se utiliza un solo flujo de datos y un flujo de instrucciones, por lo tanto, los programas creados solo permiten realizar una sola tarea o proceso. Los hilos permiten ejecutar más de un proceso de forma simultánea, de esta forma se puede aprovechar y optimizar de mejor manera el uso del procesador para nuestros programas y hacer que puedan responder a más de dos tareas a la vez.
* Las aplicaciones en el mundo real deben realizar mas de dos procesos a la vez, como descargar un archivo, recibir mensajes o notificaciones, redactar un mensaje, actualizar una lista de noticias, etc. Gracias a los hilos se puede gestionar y definir varios procesos, pero tomando en cuenta su comportamiento para evitar errores o fallos en el sistema.

# RECOMENDACIONES

* Se debe considerar que al tener dos procesos que dependan uno del otro puede provocar inconsistencias o fallos si no se controla la forma y el orden en el que se ejecutan, para esto se deben manejar conceptos de sincronización e interrupciones.
* Existen varios lenguajes de programación, si el proyecto que se esta realizando necesita manejar varios procesos a la vez hay que considerar que algunos lenguajes como JavaScript no poseen la capacidad de crear ni gestionar hilos.

# REFERENCIAS

Blancarte, O. (29 de Marzo de 2017). *Concurrencia vs Paralelismo*. Obtenido de oscarblancarteblog: https://www.oscarblancarteblog.com/2017/03/29/concurrencia-vs-paralelismo/

Díaz, J. (2022). *Píldoras Informáticas*. Obtenido de YouTube: https://www.youtube.com/playlist?list=PLU8oAlHdN5BktAXdEVCLUYzvDyqRQJ2lk

Nakayama, A. (2009). Guía práctica de estudio 12: Hilos. México.