DOCUMENTACIÓN
CARRERA DE COCHES EN JAVAFX

Eder Martínez Castro

20 de noviembre de 2025

Índice general

1. Análisis y Especificación de Requisitos

1.1. Descripción del problema

Se nos solicita crear un **programa en Java** que simule una **carrera de coches** usando **hilos**, aparte de crearle una interfaz gráfica utilizando `JavaFX`. Cada coche se ejecutará en un hilo independiente y compite contra los demás para poder llegar a la meta lo antes posible.

- Cada **coche** tiene un `nombre`, una misma `distancia total` a recorrer y una `velocidad máxima`.
- El avance para cada coche será **aleatorio** dentro de su velocidad máxima para así poder simular cambios de velocidad durante la carrera.
- Cada coche se moverá de manera **concurrente** y actualizatá su posición en nuestra interfaz.
- Cuando un coche cruce la meta, se registrará su **tiempo de llegada** y se actualizará la **clasificación**.

1.2. Requisitos funcionales

A continuación, se listan los requisitos funcionales del sistema:

ID	Descripción
RF-01	**Crear los coches**: cada coche tiene `nombre:String`, `finishDistance:int` y `speedMax:int`.
RF-02	**Hilos de carrera**: cada coche se ejecutará en un `Thread`, para simular la carrera.
RF-03	**Avance aleatorio**: el coche avanzará una distancia de manera aleatoria entre 0 y `speedMax`.
RF-04	**Actualizar la interfaz**: la posición de cada coche se mostrará en la interfaz mediante el movimiento de su `ImageView`.
RF-05	**Registro de llegada**: cuando un coche cruza la meta, se llamará al método `finish()` del controlador para registrar su tiempo y actualizar la clasificación.
RF-06	**Clasificación en tiempo real**: la interfaz mostrará el orden de llegada de todos los coches y cuántos faltan por terminar.

1.3. Requisitos no funcionales

A continuación, se listan los requisitos no funcionales del sistema:

ID	Descripción
RNF-01	**Lenguaje**: el programa se desarrollo en `Java` usando `JavaFX` para crear la interfaz.
RNF-02	**Experiencia de usuario**: crear una interfaz simple, clara y bonita, para mostrar la carrera.

2. Arquitectura del sistema

2.1. Arquitectura general

La arquitectura de está aplicación se explicará en base a lo realizado en `JavaFX` ya que sigue una arquitectura **Modelo–Vista–Controlador (MVC)**.

- **Modelo:** Tenemos la clase `Car` que representa el comportamiento de cada coche durante la carrera. Está tiene la lógica para el avance, el cálculo del tiempo de llegada a la meta y la comunicación con nuestro controlador.
- **Vista:** Nuestra interfaz la tenenmos definida en el `index.fxml` y su hoja de estilos `index.css`. Muestra la pista de carreras, los coches mediante imágenes y la clasificación final de la carrera.
- **Controlador:** Tenemos la clase `AppController` que nos permite coordinar la carrera: inicia los hilos, actualiza las posiciones de los coches y registra el orden de llegada a la meta.

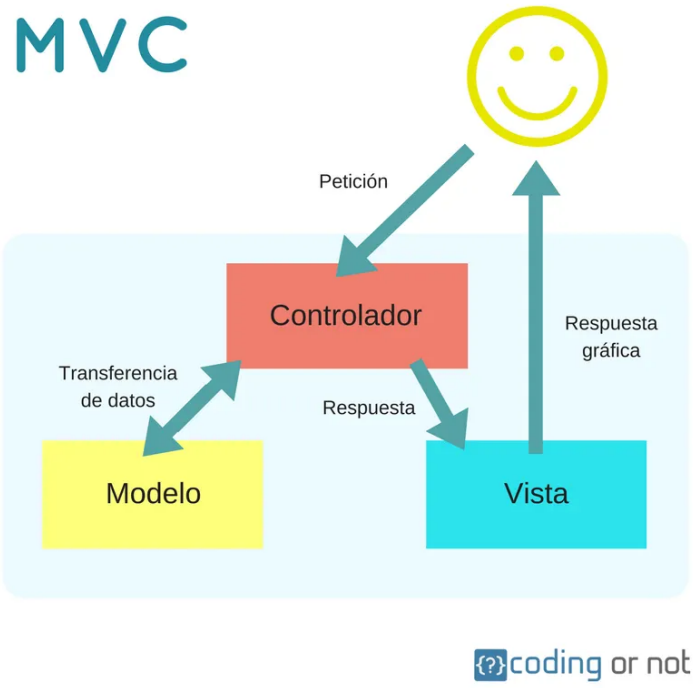

Figura 1: Arquitectura MVC

2.2. Stack empleado

Para el desarrollo de nuestra aplicación hemos utilizado las siguientes tecnologías:

- `Java` como **lenguaje de programación** y entorno de ejecución.
- `JavaFX` para crear la **interfaz gráfica** de nuestra aplicación.
- `CSS` para dar estilos y mejorar la estética a nuestra interfaz.

2.3. Interfaz

Para nuestra interfaz gráfica empleamos los siguientes elementos:

- El **título** de nuestra aplicación.
- Un **botón** para poder iniciar la carrera.
- Una **pista de carreras** con los coches colocados en sus puestos de salida y la línea de meta.
- Una **clasificación** donde se verá quien ha cruzado ya la meta.

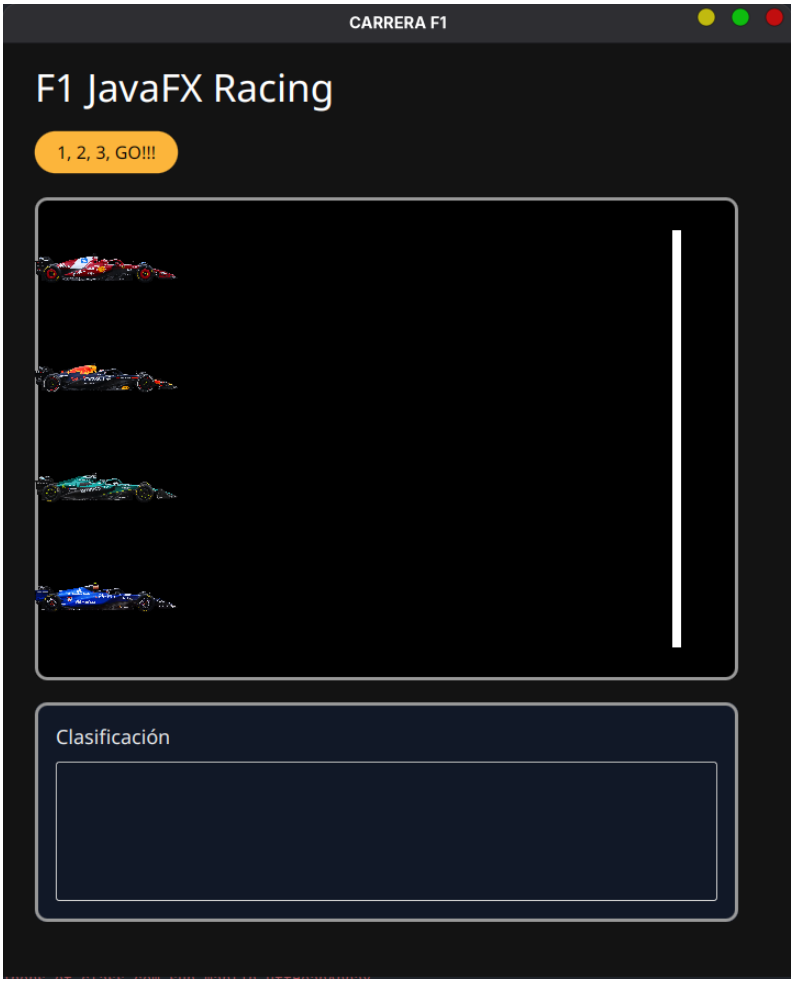

Figura 2: Interfaz de la carrera en JavaFX

3. Gestión de hilos y uso de `synchronized`

3.1. Concurrencia en la aplicación

En nuestra aplicación de `JavaFX`, cada coche se ejecuta en un hilo (la clase `Car` hereda de `Thread`). Pero todos estos hilos comparten el mismo `AppController`, que es el controlador de la interfaz.

Cuando un coche cruza la línea de meta, este llama al siguiente método:

```
public synchronized void finish(String pilotName, long finishNano
   ) {
    pilots.add(pilotName);
    times.add(finishNano);

    // Ordenar por tiempo de llegada
    List<Integer> index = new ArrayList<>();
    for (int i = 0; i < times.size(); i++) {
        index.add(i);
    }
    index.sort(Comparator.comparingLong(times::get));

    // Crear la clasificacion
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < index.size(); i++) {
        int position = index.get(i);
        String pilot = pilots.get(position);
        sb.append("#").append(position + 1)
          .append(". ").append(pilot).append("\n");
    }

    // Comprobar si faltan coches
    if (pilots.size() < 4) {
        int restantes = 4 - pilots.size();
        sb.append("\nFaltan por llegar ")
          .append(restantes).append(" coche(s)...");
    } else {
        raceRunning = false;
    }

    // Actualizar interfaz
    Platform.runLater(() -> {
        classification.setText(sb.toString());
        if (!raceRunning && startButton != null) {
            startButton.setDisable(false);
        }
    });
}
```

3.2. Por qué utilizamos `synchronized` en `finish()`

Si no usamos `synchronized`, nos podrían succeder los siguientes problemas:

- Los coches podrían cruzar la meta casi al mismo tiempo.
- Cada hilo estaría llamando a `finish()` de manera simultánea.
- Todos los hilos intentarían modificar al mismo tiempo las listas `pilots` y `times`, provocando estos fallos:
  - Podrían perderse o eliminarse.
  - El orden de llegada podría mostrarse de manera érronea.
  - El nº de coches que faltan por llegar también estaría mal calculado.

Como usamos `synchronized` en nuestro método `finish()` este nos asegura:

- Qué cada coche esperará su turno para poder registrar cuando cruza la meta.
- Se actualizarán las listas y se calculará la clasificación de manera correcta.
- Se mostrará en la interfaz el orden en el que cruzarón la meta de manera y el nº de coches que faltan por llegar de manera correcta.

En conclusión, Utilizamos `synchronized` para poder **proteger cuando actualizamos** el estado de la carrera, evitando así que varios hilos puedan chocar entre sí y asegurando que el resultado que mostremos sea el correcto.