	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): René Adrián Dávila Pérez

Asignatura: Programación Orientada a Objetos

Grupo: 1

No de Práctica(s): 111213

Integrante(s): 322184022

322238682

321570789

322069093

322249723

No. de lista o brigada: 5

Semestre: 2026-1

Fecha de entrega: 26/09/2025

Observaciones:

CALIFICACIÓN: _____

Índice

1. Introducción

1.0.1. Planteamiento del problema

La práctica se enfoca en analizar el funcionamiento de distintos ejemplos relacioandos con el manejo de archivos y la ejecución de tareas concurrentes mediante hilos. Se da conocer como se administran las tareas asíncornas, como se coordina el trabajo paralelo y el uso de archivos. Además, se busca identificar como se integran en un programa real, observar su funcionamiento y aplicación.

1.0.2. Motivación

La motivación de esta práctica es entender cómo se comporta una aplicación real cuando incorpora conceptos como el Event Loop, async/await, Future, isolates y comunicación mediante puertos. También la importancia de la concurrencia y del uso de archivos para el manejo de información, obteniendo una mejor visión de su funcionamiento, comprendiendo estos conceptos junto con su implementación.

1.0.3. Objetivos

- Analizar el comportamiento de las tareas asíncronas
- Explicar el funcionamiento de isolates
- Identificar la función del manejo de archivos

2. Marco Teórico

2.1. Manejo de Archivos

[2]

- **Librería dart.io:**
  Es el paquete estándar en Dart diseñado para trabajar con Entrada/Salida (I/O) en aplicaciones que corren fuera del navegador. Su propósito principal es proveer APIs para interactuar con el sistema operativo, incluyendo archivos, sockets, procesos y servidores HTTP.

- **Clase File:**
  La clase File dentro de la librería *dart.io* es la abstracción que representa un archivo en el sistema de archivos local. Se instancia pasando la ruta del archivo *(path)*:

- **Lectura y Escritura de Archivos:**

  - **Métodos síncronos:**
    Son la forma estándar en Dart. No bloquean el hilo principal. Cuando se llama a una función asíncrona, el Event Loop continúa procesando otros

eventos mientras el disco duro realiza la operación. *Retornan un objeto Future. Ejemplos: readAsString, writeAsString*

- **Métodos asíncronos:**
  Bloquean la ejecución del programa hasta que la operación de archivo termina. *Ejemplos: readAsStringSync, writeAsStringSync*

- **Manejo de Excepciones**
  El manejo de archivos es propenso a errores externos (archivo inexistente, disco lleno, falta de permisos). En Dart, esto se gestiona mediante bloques *try-catch* capturando excepciones específicas como *FileSystemException*

2.2. Hilos

[1]

- **Modelo de Concurrencia de Dart**
  A diferencia de lenguajes como Java que utilizan múltiples hilos de sistema por defecto para la concurrencia, Dart es, en su núcleo, **Single-Threaded** (de hilo único). El funcionamiento se basa en el Event Loop (Bucle de Eventos). El hilo principal ejecuta el código síncrono *línea por línea*. Cuando encuentra tareas asíncronas, no se detiene a esperar, delega la tarea y continúa ejecutando. Cuando la tarea externa termina, coloca el resultado en una cola de eventos para ser procesado cuando el hilo principal esté libre.

  - **Bloqueo (Blocking):**
    Ocurre cuando una operación tarda mucho tiempo en completarse y ocupa el hilo principal, impidiendo que el Event Loop procese otros eventos (como clicks de usuario)
  - **No Bloqueo (Non-Blocking):**
    Permite iniciar una tarea y retornar el control inmediatamente al Event Loop

  Asincronía:

- **Future:**
  Es un objeto que representa un valor potencial o un error que estará disponible en algún momento en el futuro.

  - **async y await**
    Son palabras clave que permiten escribir código asíncrono que se "leeçomo si fuera síncrono. *await* pausa la ejecución de la función actual hasta que el Future se completa, pero sin bloquear el hilo principal.
  - **Future.delayed**
    Es un constructor que crea un *Future* que se completa después de un tiempo específico, útil para simular retardos de red o programar tareas.

Concurrencia: [3]
Cuando se requiere procesamiento computacional pesado (CPU-bound) que bloquearía el Event Loop (como procesar una imagen grande o cálculos matemáticos complejos), la asincronía simple no es suficiente. Aquí entra el Paralelismo Genuino mediante **Isolates**.

- **Isolate:**
  Un Isolate es un hilo de ejecución independiente. A diferencia de los hilos en Java o C++, los Isolates *no comparten memoria*. Cada Isolate tiene su propia pila de memoria (heap) y su propio Event Loop

- **Modelo de Comunicación:**
  Dado que no comparten memoria, no existen problemas de çondición de carrera"(race conditions) típicos de los hilos compartidos. La comunicación se realiza exclusivamente enviando mensajes a través de puertos:
  SendPort: Se utiliza para enviar mensajes a un ReceivePort específico
  ReceivePort: Escucha los mensajes entrantes.

2.3. Patrones de Diseño

- **Patrón Modelo-Vista-Controlador (MVC)**
  [5] Es un patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario y la lógica de control en tres componentes distintos:
  Modelo: Gestiona los datos, la lógica de negocio y las reglas de la base de datos.
  Vista (View): Es la representación visual de la información (lo que el usuario ve). **Controlador:** Intermediario que recibe las entradas del usuario (desde la Vista), las procesa (usando el Modelo) y actualiza la Vista.

  - **Propósito:**
    Su objetivo principal es la Separación de Preocupaciones (Separation of Concerns). Esto permite que el código sea más modular, facilitando que un equipo trabaje en la interfaz gráfica mientras otro trabaja en la lógica de datos sin interferirse, mejorando el mantenimiento

- **Patrón Singleton:** El Singleton es un patrón de diseño creacional [4]

  - **Propósito:**
    Garantizar que una clase tenga **una única instancia** y proporcionar un punto de acceso global a ella.
  - **Casos de Uso Comunes:** Gestión de conexiones a bases de datos, sistemas de configuración global, o servicios de registro (logging)

3. Desarrollo

3.1. Códigos de Archivos

1. **Uso de la Librería de Entrada/Salida (dart:io)**
   El programa inicia importando la librería dart:io, que contiene las clases *File*, *stdin* y *stdout* necesarias para interactuar con el sistema operativo y gestionar los flujos de datos

   - **Entrada (stdin):** Se utiliza *stdin.readLineSync()* para capturar la información que el usuario escribe en la consola (nombre del archivo, contenido del texto, confirmaciones)
   - Salida (stdout): Se emplea *stdout.write* para mostrar los mensajes del menú y las instrucciones sin saltos de línea automáticos

2. **Instanciación del Archivo**
   En las funciones crearYEscribirArchivo, leerArchivoExistente y sobrescribirArchivo, se aplica el concepto de instanciación de la clase File:

   ```
   File archivo = File(nombreArchivo)
   ```

   Esto crea un *objeto* que representa la ruta en el sistema de archivos, aunque el archivo físico podría no existir aún en ese momento

3. **Operaciones de Escritura Síncrona**
   Para las opciones de "Crear" y "Sobrescribir", el código utiliza el método writeAsStringSync.
   Aplicación en el código:

   ```
   archivo.writeAsStringSync(lineas.join('\n'));
   ```

   "Syncïndica que la operación es síncrona, significa que el flujo del programa se detiene y espera a que la escritura se complete antes de continuar con la siguiente línea de código
   Antes de escribir, el código utiliza *lineas.join()*., eso une los elementos de la lista en una sola cadena, conservando los saltos de línea que el usuario puso
   Si el archivo no existe, este método lo crea; si ya existe, reemplaza todo su contenido anterior.

4. **Operaciones de Lectura**
   En la función $_{i}eerArchivoExistente$, se aplican:

   - $if(!archivo.existsSync())$: Comprueba si el archivo existe antes de intentar leerlo, evitando errores de ejecución.

- *archivo.readAsStringSync*(): Lee todo el contenido binario del archivo y lo convierte a una cadena de texto (String) para poder imprimirlo en consola

5. **Manejo de Excepciones en Archivos** El manejo de archivos es propenso a errores externos (rutas inválidas, permisos denegados, disco lleno). El código implementa bloques try-catch en sus operaciones:

```
try {
    // Operaciones con File
} catch (e) {
    print('Error: $e');
}
```

   Esto nos permite asegurar que si ocurre una falla o excepcion durante la interacción con el archivo, el programa no se cierre, sino que capture el error y muestre un mensaje al usuario

6. Sobrescritura En la función $_s$*obrescribirArchivo*, se hace una validación antes de llamar el método de escritura, se solicita una confirmación ("SI") al usuario, esto para evitar la pérdida de datos.

3.2. Códigos de Hilos

1. El ejemplo muestra como con el método **Future** se programa una tarea que es imprimir: "Tarea asíncrona completada", para que se ejecute después de 2 segundos. Esto se ejecuta en Event Loop, ya que es la manera en la que Dart maneja tareas concurrentes, y todo funciona en un solo hilo, las tareas son eventos concurrentes porque se ejecutan de forma asíncrona.

2. La única diferencia está en que el método Future implementa un **await** (hace que el código espere por el resultado de Future antes de pasar a la siguiente línea), entonces da la indicación de que la tarea se suspenda esperando a que Future se complete, es decir, cuando pasen 2 segundos. Pasando este tiempo la ejecución se reanuda y se imprime: "Tarea completada con await".

3. La parte de **port.send** es el puerto de envío que el nuevo Isolate usa para enviar mensajes de vuelta al **receivePort** que es el puerto de escucha donde se reciben los mensajes. Entonces la función main sea un receivePort, crea un Isolate llamando a **Isolate.spawn**, y la función tarea comienza a ejecutarse en un hilo paralelo, recordando que los Isolate ejecutan código en hilos de ejecución separados, y que no comparten memoria, solo se comunican a través de mensajes. Posteriormente la tarea envía el mensaje "Hola desde otro isolate" de vuelta al hilo principal usando el sendPort, finalmente, el receivePort recibe el mensaje, lo imprime y cierra el puerto.

Es importante marcar el fin del proceso *receivePort.close()*, si no después de imprimir se mantiene el modo listen y el isolate seguiría en ejecución tomando el control del sistema operativo esperando ”algo”.

4. La clave de este código es que se ejecutan tareas pesadas de CPU en un isolate para mantener el hilo principal libre.
   La función **sumaGrande** es el trabajo pesado, y en el main se crea un *receivePort*, se imprime el mensaje ”Iniciando tarea pesada en hilo paralelo...”, con el método **Isolate.spawn** se comienza a ejecutar el bucle en paralelo, inmediatamente después de imprime el segundo mensaje, cuando el isolate termina de sumaGrande envía el resultado por el puerto de envío y el puerto que recibe en el hilo principal recibe el resultado, lo imprime y cierra el puerto.

5. Primero el main crea su receivePort y le pasa su sendPort al ”worker” al hacer Isolate.spawn, mainPort es un tipo de envio de puerto, que va a apuntar al método main, y el receive prepara el método para recibir el mensaje desde main workerSendPort, este es el puerto de envío que el main usará para comunicarse (enviar el mensaje real) con el worker, finalmente, se cierra el puerto.
   En resumen, se muestra la comunicación entre un isolate principal (main) y un isolate secundario (worker), los cuales permiten el intercambio de mensajes en ambas direcciones.

6. Primero se imprime ”inicio”, luego ejecuta inmediatamente un bucle for muy grande para calcular una suma. Debido a que esta tarea es síncrona, el hilo principal de Dart queda bloqueado (congelado) hasta que el bucle se completa, lo que puede tardar varios segundos, después de que se termina el cálculo, el programa imprime el resultado e imprime ”Fin”. El principal inconveniente es que bloquea el Event Loop, impidiendo que el programa responda a cualquier otro evento o actualice la interfaz de usuario durante la duración de la tarea pesada.

3.3. Códigos de Patrones

3.3.1. Patrón MVC

Archivo *main.dart* de Patrones.
Según nuestro marco teórico, el MVC busca la 'Separación de Preocupaciones' dividiendo la aplicación en 3 componentes.
En el código de Pokémon se aplica así:

1. **Modelo (*Pokemon, Ataque*)**:
   Gestiona los datos y la lógica de negocio.
   Las clases *Pokemon* y sus herencias (*PokemonFuego, PokemonHierba*) definen la estructura de los datos (vida, velocidad, tipo). No saben nada de cómo se muestran en pantalla, simplemente almacenan el estado.

2. **Vista (*CombateView, ConsoleCombateView*)**:
   Es la representación visual de la información
   La clase abstract y su implementación *ConsoleCombateView* contienen solamente print(). Son para mostrar qué está pasando (ataques, daño, ganador), pero no decide nada sobre el combate

3. **Controlador (*CombateController*):** procesa la lógica y actualiza la vista
   La clase CombateController contiene el bucle while del juego.
   Toma los datos del Modelo (velocidad de los Pokémon para decidir quién ataca)
   Calcula la lógica (daño, multiplicadores de tipo)
   Ordena a la Vista que muestre los resultados (view.mostrarAtaque, view.mostrarGanador)

3.3.2. Patrón Singleton

Archivo *Singleton.dart* de Patrones.
El objetivo de este patrón es garantizar una única instancia de una clase y proveer un punto de acceso global. [4] La clase debe mantener una instancia privada y estática.
Código:

```
static final Impresora _instancia = Impresora._interna();
```

Al ser static, pertenece a la clase y no al objeto, asegurando que sea única en memoria.
Se debe mantener el constructor privado para evitar que clientes externos creen nuevas instancias con *new*.
Código:

```
Impresora._interna();
```

es un constructor nombrado privado que impide hacer instancias.
También debe tener un método público y estático para acceder a la instancia
Código:

```
factory Impresora() { return _instancia; }
```

Permite que cuando el usuario llame a *Impresora()*, el sistema le devuelva la instancia ya creada en lugar de una nueva

3.4. Diagramas UML

A continuacion se muestran las representaciones de la estructura de los programas en diagramas UML estatico (clases) y dinamico (secuencia).

3.4.1. Archivos

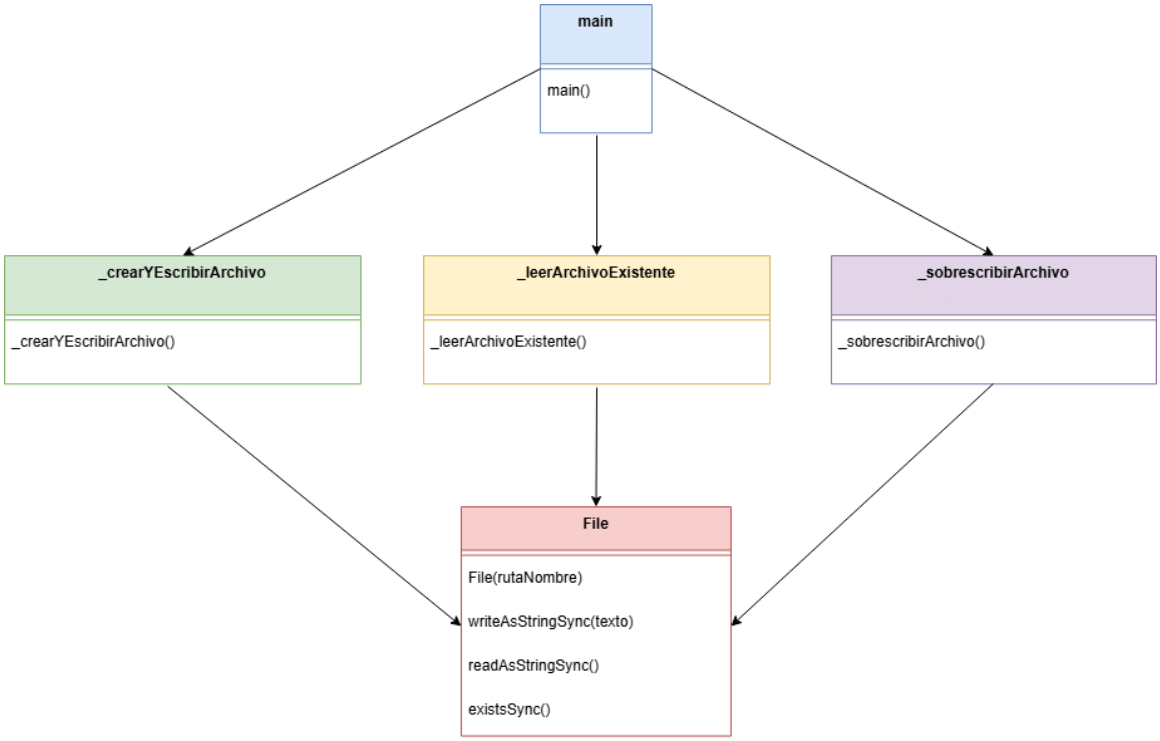


Figura 1: Diagrama de clases archivos

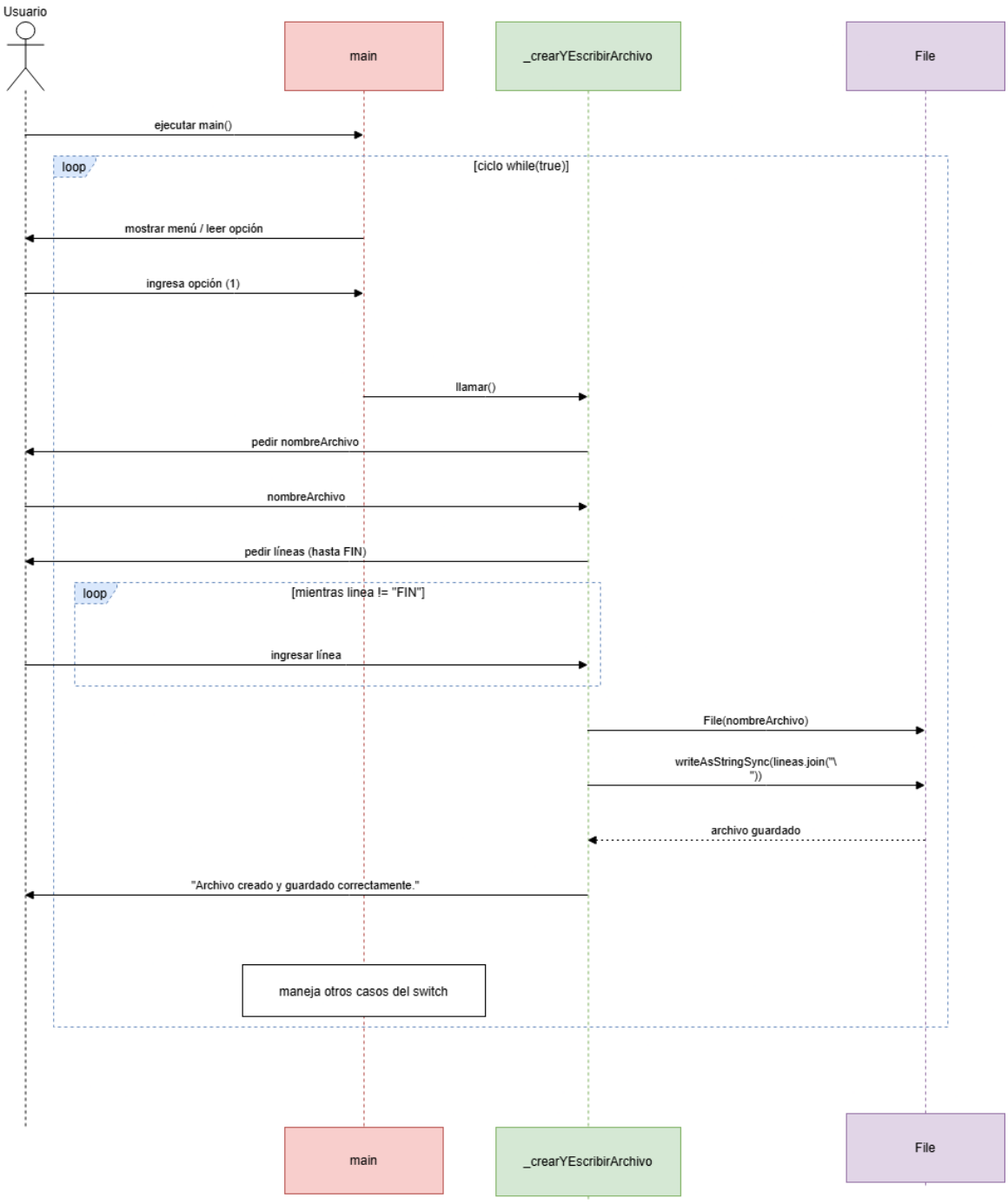


Figura 2: Diagrama de secuencia archivos

3.4.2. Hilos

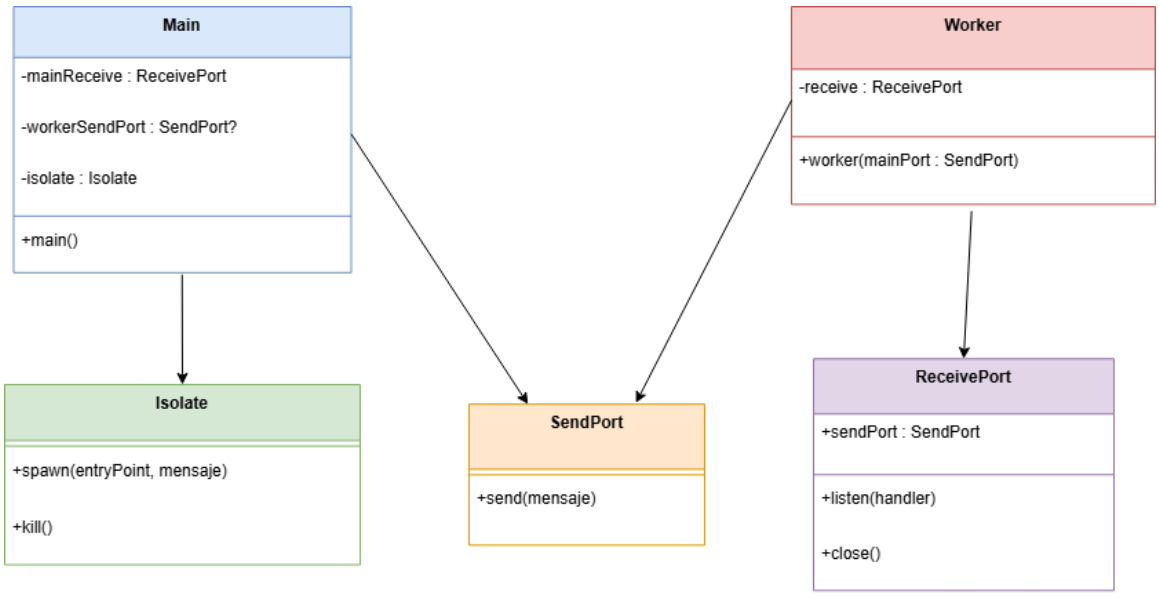


Figura 3: Diagrama de Clases Hilos

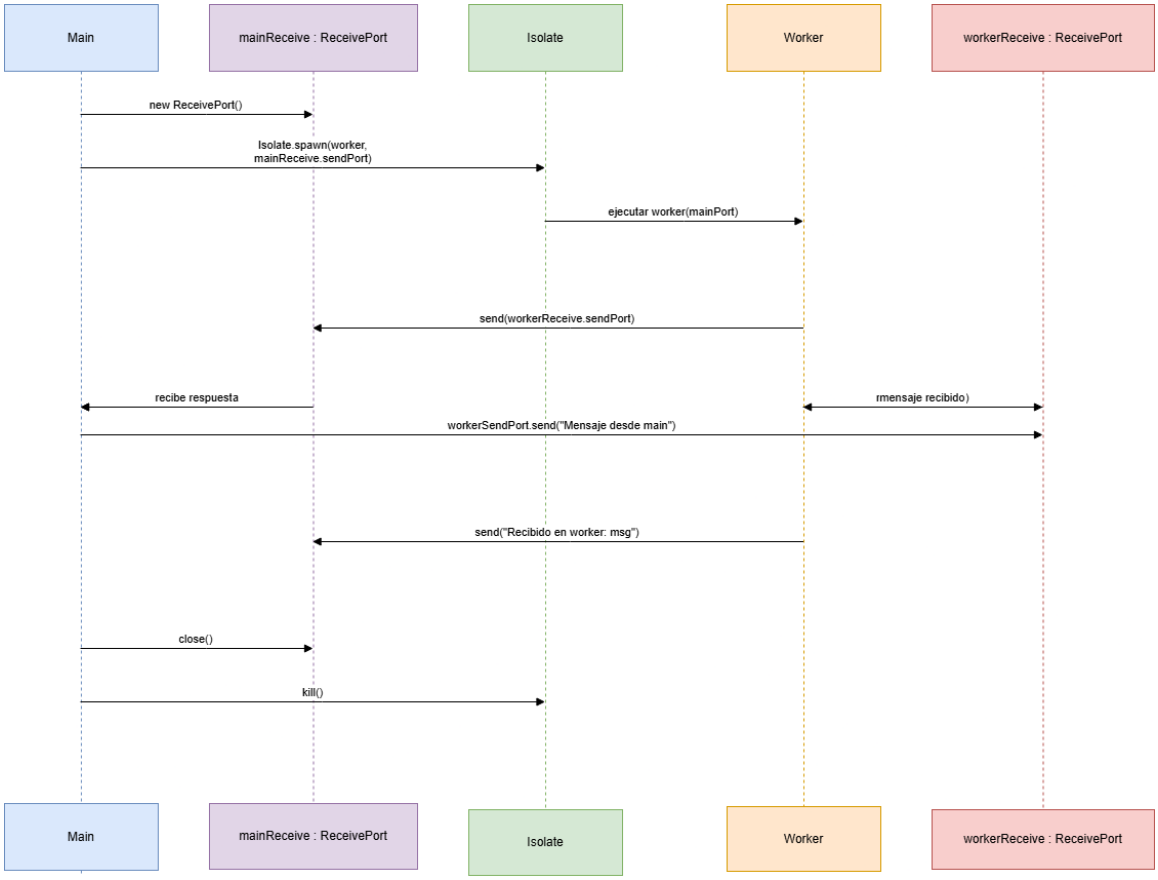


Figura 4: Diagrama de Secuencia Hilos

3.4.3. Patrones

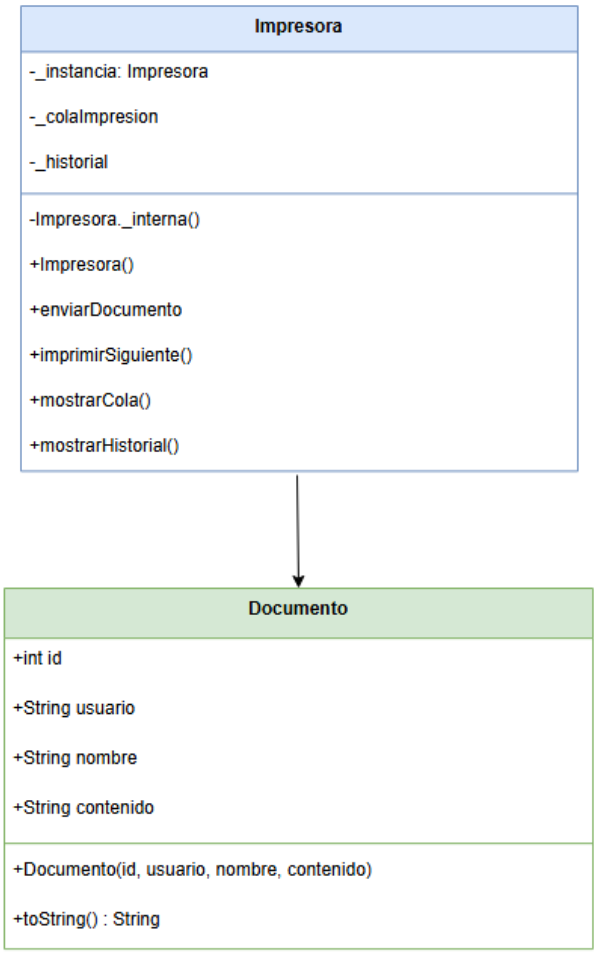


Figura 5: Diagrama de Clases Singleton

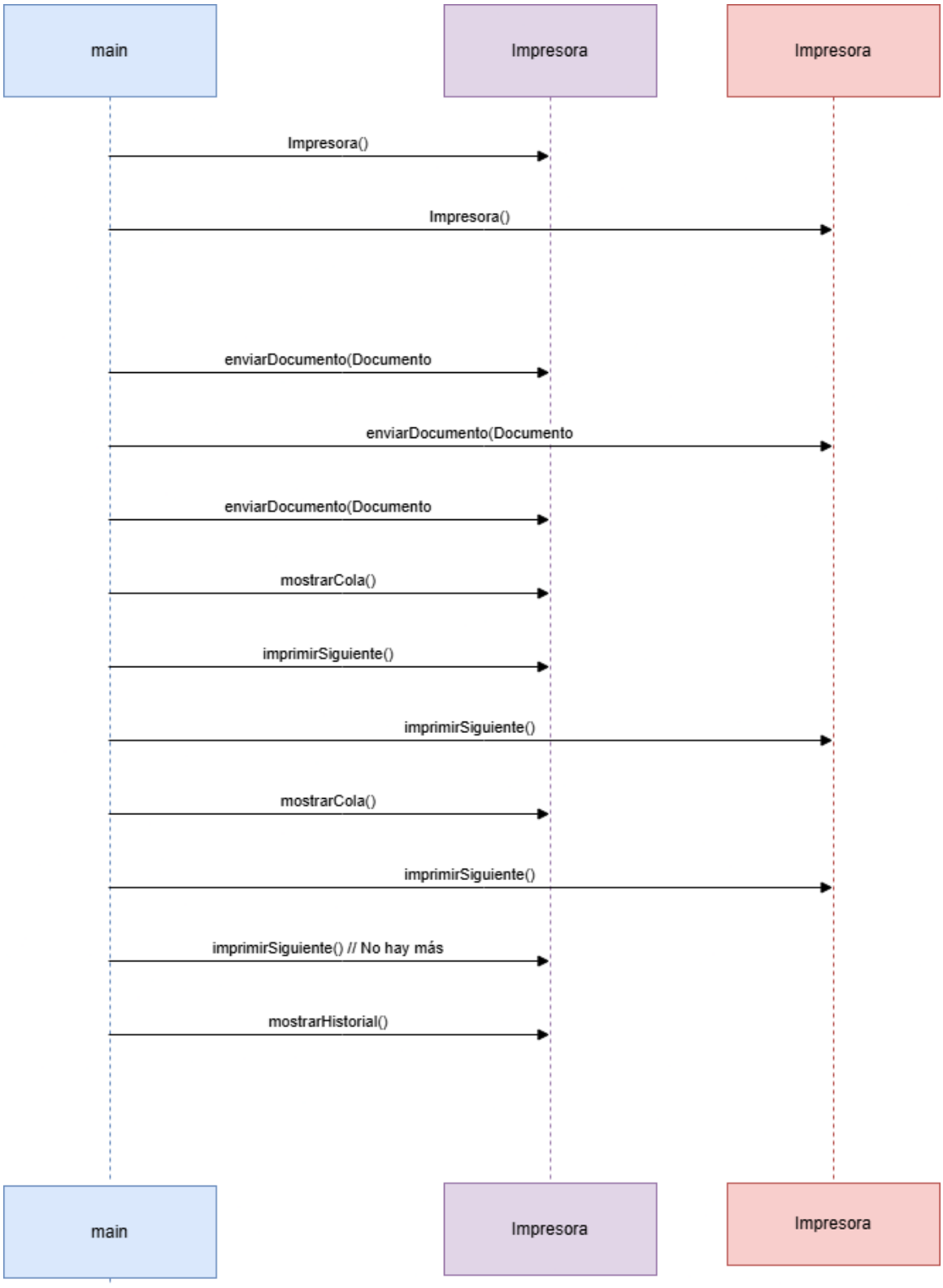


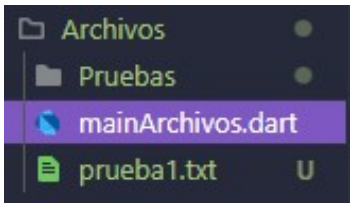
Figura 6: Diagrama de secuencia Patrón Singleton

4. Resultados

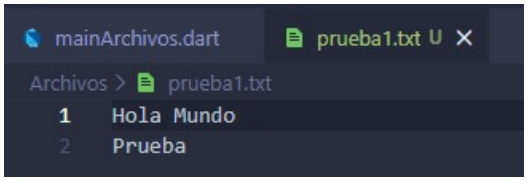
4.1. Archivos

```
roman@RomL MINGW64 ~/POO/POO/Practica111213/Archivos (main)
$ dart mainArchivos.dart
============================
       MENÚ DE ARCHIVOS
============================
1) Crear archivo .txt y escribir texto
2) Leer archivo existente
3) Sobrescribir archivo existente
4) Salir
Elige una opción: 1
Nombre del archivo a crear (ej: notas.txt): prueba1.txt

Escribe el texto que deseas guardar.
Para terminar, escribe SOLO: FIN
---------------------------------
Hola Mundo
Prueba
FIN

Archivo creado y guardado correctamente.

============================
       MENÚ DE ARCHIVOS
```

Figura 7: Creación de Archivo y su contenido 'prueba1.txt'



(a) Archivo creado en la misma carpeta que el código fuente



(b) Contenido dentro del archivo

Figura 8: Comprobación de la creación y escritura del archivo con los datos ingresados

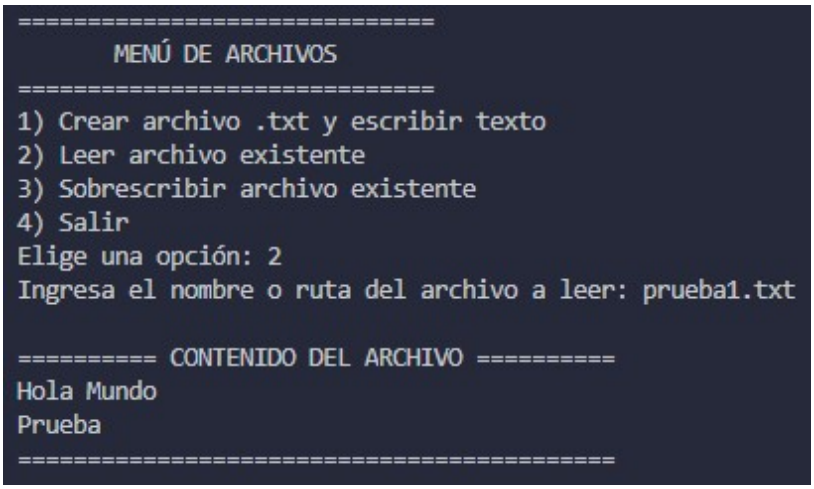


Figura 9: Impresión en consola del contenido del archivo

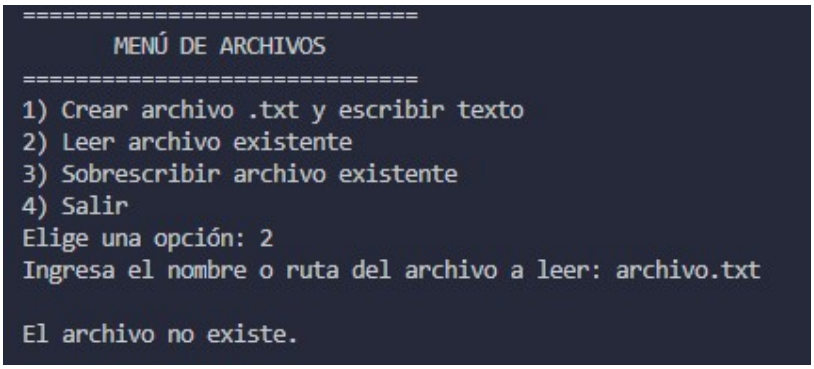


Figura 10: Prueba donde se ve el manejo de excepciones para el caso de intentar sobreescribir un archivo no existente

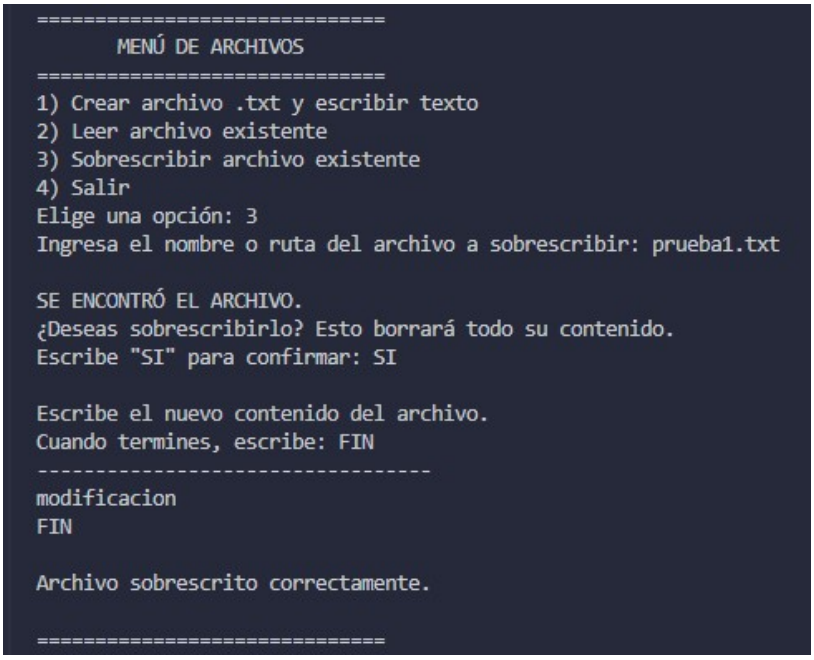


Figura 11: Sobreescritura de un archivo existente

Figura 12: Comprobación de sobreescritura desde el archivo

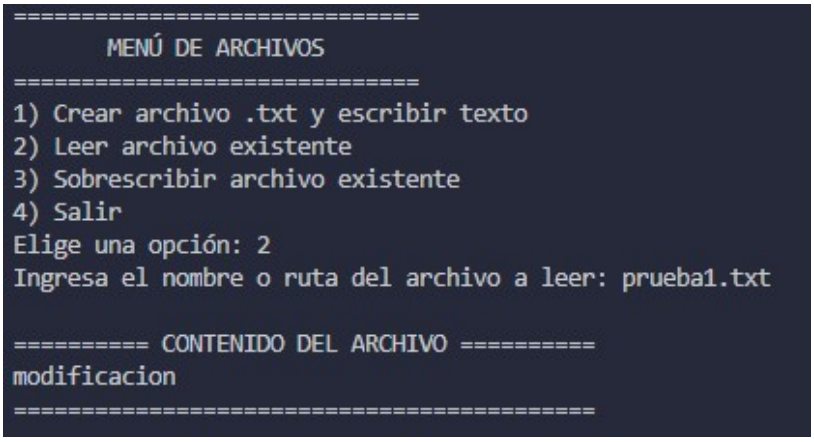


Figura 13: Impresión en consola del contenido sobreescrito del archivo

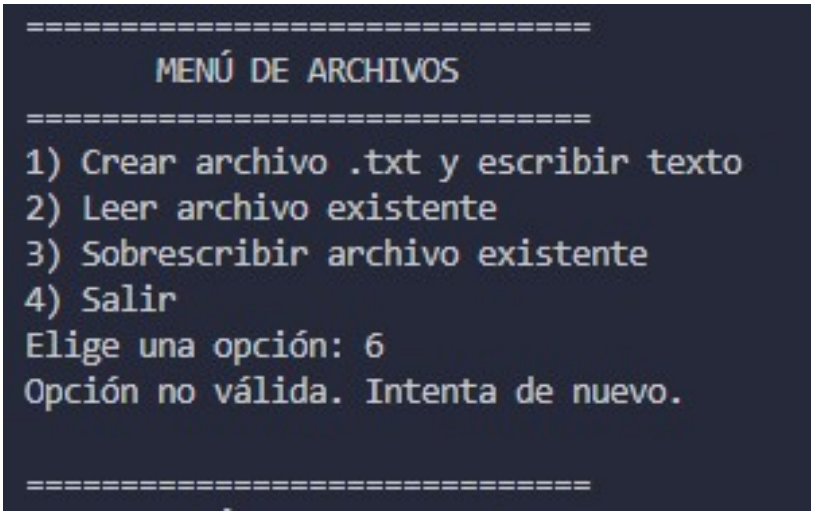


Figura 14: Prueba de opción no válida en el menú

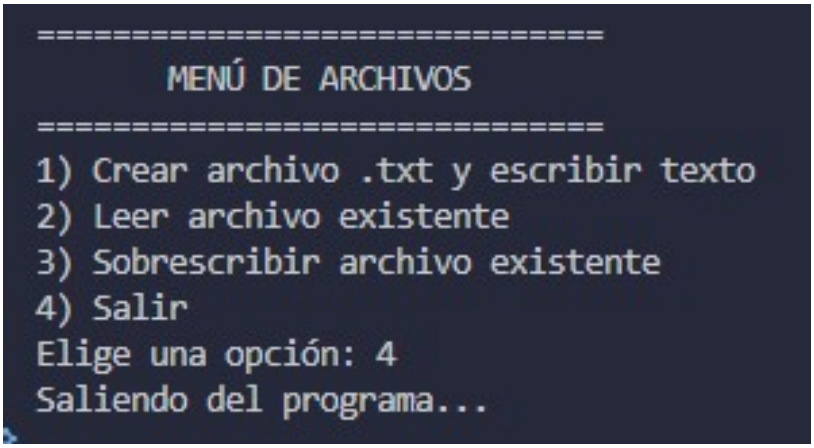


Figura 15: Salida

4.2. Hilos

```
Future.delayed(Duration(seconds: 2), () {
  print("Tarea asíncrona completada");
});
```

Figura 16: Ejemplo de Future sin await

```
await Future.delayed(Duration(seconds: 2));
print("Tarea completada con await");
```

Figura 17: Ejemplo de Future con await

```dart
void main() async {
  final receivePort = ReceivePort();

  await Isolate.spawn(tarea, receivePort.sendPort);

  receivePort.listen((mensaje) {
    print(mensaje);
    receivePort.close();
```

Figura 18: Isolate enviando mensaje al hilo principal

```
Iniciando tarea pesada en hilo paralelo...
Mientras tanto, sigo ejecutando en el hilo principal...
Resultado: 124999999750000000
```

Figura 19: Isolate ejecutando tarea pesada en paralelo

```
Main: recibí el SendPort del worker.
Main: respuesta del worker -> Recibido en worker: Mensaje desde main
```

Figura 20: Comunicación entre isolate principal y worker

```
Inicio
Resultado: 124999999750000000
Fin
```

Figura 21: Ejecución síncrona bloqueando el hilo principal

4.3. Patrones

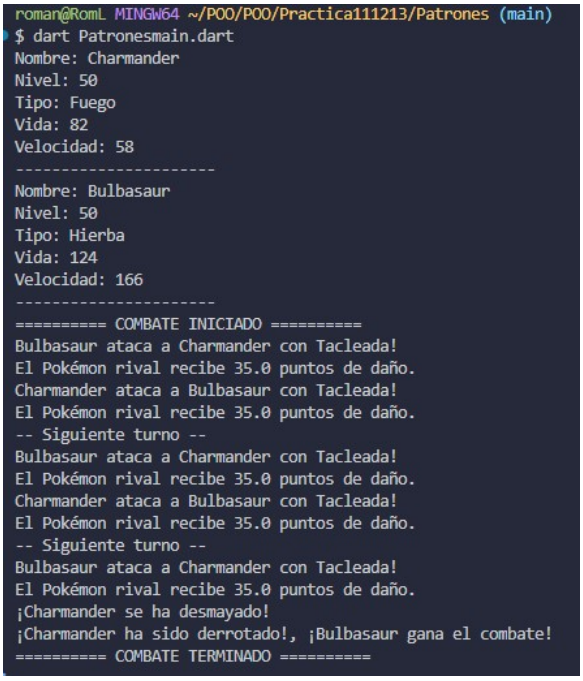


Figura 22: Simulación de Combate (MVC)

Explicación:

1. *iniciarCombate* recibe a Charmander y Bulbasaur: La Vista muestra el estado inicial de los Modelos
2. Controlador compara velocidades: Lógica de negocio (Controlador, cálculo interno)
3. atacar: Charmander usa Tacleada, la vista notifica la acción
4. Cálculo de daño: Controlador actualiza estado del Modelo
5. Bulbasaur sigue vivo (vida >0), entonces contraataca
6. El bucle continúa hasta que vida <= 0
7. Vida de Bulbasaur llega a 0: El Controlador detecta la condición de fin y la Vista muestra el resultado

4.3.1. Singleton

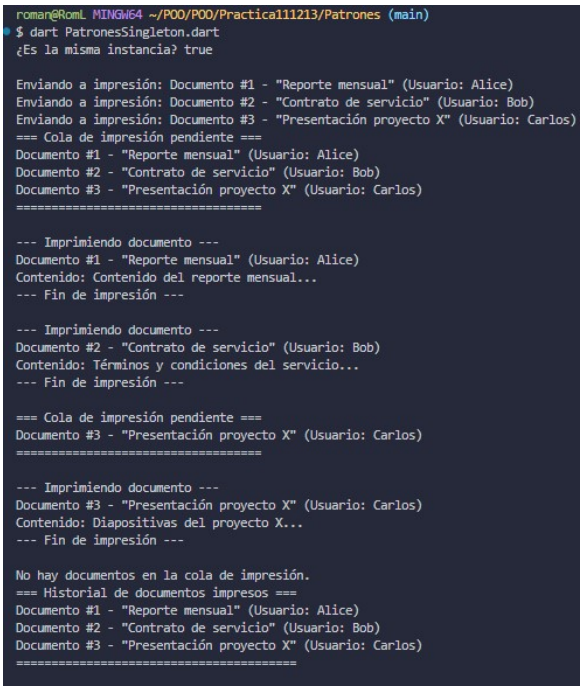


Figura 23: Prueba de Singleton

Se debe demostrar que dos variables distintas apuntan al mismo objeto en memoria:

1. ```
   final impresoraA = Impresora();
   final impresoraB = Impresora();
   ```

   : Se obtienen referencias a **la misma** instancia

2. $print(identical(impresoraA, impresoraB))$ Confirma que no se crearon dos objetos diferentes

3. $impresoraA.enviarDocumento(...)$: Se agrega a la cola de la instancia única

4. $impresoraB.enviarDocumento(...)$: Se agrega a la misma cola que A, aunque usó variable impresoraB

5. $impresoraA.mostrarCola()$: Confirma que ambos documentos están en el mismo recurso compartido

6. $impresoraB.imprimirSiguiente$ :: impresoraB puede procesar el documento que envió impresoraA
   ```

5. Conclusiones

La práctica cumple los objetivos, ya que se logró analizar el comportamiento de las tareas asíncronas y la forma en que el lenguaje organiza su ejecución mediante Future, async/await y el Event Loop, permitiendo así entender cómo se manejan operaciones que requieren esperar sin bloquear el flujo principal del programa. También se explicó el funcionamiento de los isolates, observando cómo se ejecutan tareas en hilos separados y cómo se comunican mediante puertos. Asimismo, se trabajó el manejo de archivos, observando su utilidad para almacenar información.

6. Referencias

[1] Google Developers. Concurrency in dart. dart language guide. (https://dart.dev/language/concurrency), 2024. Consultado el 27-11-2025.

[2] Google Developers. Dart: Librería dart:io. dart api reference. (https://api.dart.dev/stable/dart-io/dart-io-library.html), 2024. Consultado el 27-11-2025.

[3] Google Developers. Concurrency and isolates. (https://docs.flutter.dev/perf/isolates), 2025. Consultado el 27-11-2025.

[4] Refactoring Guru. Patrón singleton. (https://refactoring.guru/es/design-patterns/singleton), s.f. Consultado el 27-11-2025.

[5] Mozilla Developer Network. Mvc (model-view-controller). mdn web docs. (https://developer.mozilla.org/es/docs/Glossary/MVC), 2023. Consultado el 27-11-2025.