| | Carátula para entrega de prácticas |
|---|---|
| Facultad de Ingeniería | Laboratorios de docencia |

# Laboratorio de Computación Salas A y B

*Profesor(a):* Dávila Pérez René Adrián

*Asignatura:* Programación Orientada a Objetos

*Grupo:* 1

*No de Práctica(s):* Practica 11, 12 y 13

*Integrante(s):* 322089020

322089817

322151194

425091586

322085390

*No. de lista o brigada:* Equipo 4

*Semestre:* 2026-1

*Fecha de entrega:* 28 de noviembre de 2025

*Observaciones:*

CALIFICACIÓN: _____

# Índice

# 1. Introducción

## 1.1. Planteamiento del Problema

El desafío en la práctica de la ingeniería de software requiere la integración de tres dominios fundamentales para la construcción de sistemas eficientes, persistentes y bien estructurados: la gestión de la información mediante Archivos, la optimización del rendimiento a través de Hilos de Ejecución, y la aplicación de soluciones de diseño probadas mediante Patrones. El problema fundamental a resolver en la formación práctica es adquirir y aplicar estos conocimientos de manera activa, trascendiendo la teoría. El objetivo es que, a partir del estudio y análisis de los códigos y ejemplos proporcionados por el profesor, el estudiante logre comprender la implementación práctica del flujo de datos en archivos, la concurrencia de tareas en hilos y la estructuración del código con patrones de diseño.

## 1.2. Motivación

El estudio de Archivos, Hilos y Patrones representa la transición de la programación básica a la ingeniería de software profesional. La motivación principal es dotar al estudiante de una base sólida para el desarrollo de sistemas complejos. Aprender sobre Archivos permite entender la persistencia y la arquitectura de un Repositorio (distinguiendo entre el Almacén Físico y Lógico). El manejo de Hilos y el Event Loop prepara para crear aplicaciones de alto rendimiento capaces de manejar múltiples tareas concurrentes. Finalmente, la comprensión de Patrones (incluyendo el Singleton y otros Patrones de Diseño/Software/Programación) proporciona el lenguaje y las estructuras para crear código mantenible, reusable y que resuelva problemas comunes de manera elegante. Este enfoque práctico garantiza que los conceptos se asimilen para ser aplicados en cualquier contexto de desarrollo.

## 1.3. Objetivos

El objetivo general de este proyecto de aprendizaje es integrar las habilidades necesarias para manejar la persistencia, concurrencia y diseño estructural del software.

Los objetivos específicos son:

- Archivos: Comprender y aplicar el concepto de fuente de datos...
- Hilos: Implementar y gestionar la concurrencia de tareas...
- Patrones: Identificar, analizar y aplicar distintos patrones de diseño...

# 2. Marco Teórico

## 2.1. Excepciones

Una excepción es un objeto en el código que durante la ejecución del programa interrumpe el flujo normal de las sentencias, es decir, se produce cuando un acontecimiento circunstancial impide el normal funcionamiento del programa. La .[ex]cepciónçontiene información sobre el acontecimiento ocurrido y transmite esta información al método desde el que se ha generado la excepción. Dependiendo del error puede cambiar la excepción, pero aun así todas las excepciones son subclases de Throwable, los tipos son los siguientes: [2]

- Checked exceptions: se deben declarar en la firma del método o capturar explícitamente en un bloque try-catch.
- Unchecked exceptions: este tipo ocurre durante la ejecución del programa y no se requiere que sean declaradas en la firma del método. Se heredan de la clase RunTimeException.
- Errors: son problemas graves están fuera de control del programador y no deben manejarse explícitamente. Estos errores terminan la ejecución del programa. [1]

El funcionamiento de las excepciones es el siguiente:

1. Definimos qué partes del programa crean una excepción y bajo qué condiciones.
2. Comprobamos si ciertas partes del programa generan una excepción.
3. En caso afirmativo se utilizan las palabras reservadas try, catch y finally. [7]

## 2.2. Archivos

Los archivos tienen como finalidad guardar datos de forma permanente. Una vez que acaba la aplicación los datos almacenados estarán disponibles para que otra aplicación pueda recuperarlos para su consulta o modificación. Las formas fundamentales de organizar un archivo son las siguientes: Secuenciales: los registros se insertan en el archivo en orden de llegada. Este tipo de archivo permite escribir, añadir al final del archivo y consultar. Directa o aleatoria: cuando un registro es directamente accesible mediante la especificación de un índice. [6]

## 2.3. Hilos

En el contexto de la ingeniería de software, un hilo (o *thread*) se define como la unidad más pequeña de procesamiento que puede ser programada por un sistema operativo. La implementación de hilos permite la **programación concurrente**, la cual es la capacidad de un programa para ejecutar múltiples tareas en periodos de tiempo superpuestos, optimizando así el uso de los recursos del CPU [5].

El uso de hilos es fundamental para mantener la capacidad de respuesta de las aplicaciones, permitiendo que operaciones pesadas se ejecuten en segundo plano. En lenguajes como Java, la concurrencia se gestiona principalmente a través de la clase `Thread` y la interfaz `Runnable`, permitiendo crear entornos multitarea eficientes y sincronizados [3].

## 2.4. Patrones de Diseño

Los patrones de diseño son soluciones probadas y documentadas para problemas recurrentes en el desarrollo de software [4]. No son fragmentos de código listos para copiar, sino plantillas o descripciones de cómo resolver un problema que pueden usarse en muchas situaciones diferentes. Su objetivo principal es facilitar la reutilización de código, mejorar la legibilidad y asegurar que el sistema sea escalable.

Estos patrones se clasifican generalmente en tres categorías: **creacionales** (manejan mecanismos de creación de objetos), **estructurales** (tratan la composición de clases y objetos) y **de comportamiento** (se ocupan de la comunicación entre objetos). La aplicación correcta de estos patrones permite desacoplar los componentes del sistema, haciendo que el software sea más robusto frente a cambios futuros.

# 3. Desarrollo

## 3.1. Análisis del Código

### 3.1.1. Archivos

Se utiliza la librería `io` para interactuar con el sistema de archivos (crear, leer, escribir) y para manejar la entrada/salida de datos. El `main` tiene un ciclo infinito que se ejecuta siempre que el usuario no seleccione salir. Se muestra un menú con 4 opciones; con `stdin.readLineSync()` se lee la entrada del teclado y siempre se convierte el texto a número con `int.tryParse`, aunque si no es un número válido, el `switch` imprime el error.

- **Opción 1 (`_crearYEscribirArchivo()`):** Crea un archivo desde cero, pide el nombre, y se pueden escribir varias líneas de texto ya que está en un ciclo el cual solo se rompe cuando se escribe "FIN". Al final se guarda con `archivo.writeAsStringSync()` y no guarda el texto con saltos de línea comunes, sino que escribe en su lugar "\n".

- **Opción 2 (`_leerArchivoExistente()`):** Esta función muestra lo que hay dentro de un archivo, pero antes de intentar leer, usa `archivo.existsSync()` para asegurarse de que el archivo exista y si no le avisa el usuario. Si el archivo está en orden, usa `archivo.readAsStringSync()` para obtener todo el texto del archivo y lo imprime en pantalla entre separadores visuales.

- **Opción 3 (`_sobrescribirArchivo()`):** Es parecida a la opción 1 pero con algunas verificaciones extra para la sobrescritura. Primero verifica que exista y obliga a escribir "SI" para confirmar; esto es muy importante porque sobrescribir borra todo el contenido anterior de archivo. Una vez que se confirma funciona igual que la opción uno, captura la información hasta escribir "FIN" y se guardan los cambios.

Cada operación está en `try` para evitar que el programa se cierre sin saber por qué.

**Interpretación** Los conceptos que abarca el programa son excepciones y archivos.

- Con los bloques `try-catch` se intenta ejecutar un bloque de código "riesgoso" (`try`) y, si algo falla, se captura el error (`catch`) para manejarlo. En el código, se utilizan envolviendo las operaciones de lectura y escritura (`writeAsStringSync`, `readAsStringSync`), ya que las operaciones de entrada/salida son impredecibles: el archivo podría estar protegido contra escritura o podría haber sido borrado.

- La manipulación de archivos se basa en la persistencia de datos mediante el sistema de archivos. Con las operaciones sincrónicas el hilo de ejecución (el programa) se "detiene" y espera a que el disco termine de leer o escribir antes de pasar a la siguiente línea de código; esto ocurre con los métodos `readLineSync`, `writeAsStringSync` y `readAsStringSync`.

### 3.1.2. Hilos

Se utiliza la librería `isolate` para la ejecución de código en paralelo y la sintaxis nativa de `Future` para el manejo de asincronía. Los archivos muestran la diferencia entre bloquear el hilo principal, usar asincronía (Futures) y usar paralelismo real (Isolates).

- En el **Ejemplo 1**, se utiliza `Future.delayed` sin esperar su finalización. El programa imprime "Inicio", agenda una tarea para dentro de 2 segundos, e inmediatamente imprime "Fin inmediato". La tarea agendada aparece al final, demostrando que el código no se bloqueó.

- En el **Ejemplo 2**, se utiliza la palabra clave `await`. Aquí, el flujo de la función `main` se pausa en la línea del `delayed`. Se imprime "Inicio", el programa espera 2 segundos reales sin ejecutar la siguiente línea, y luego imprime "Tarea completada" y finalmente "Fin".

- El **Ejemplo 6** muestra un error común: ejecutar una tarea pesada (un ciclo de 500 millones de iteraciones) directamente en el `main`. Esto congela la ejecución hasta que el ciclo termina, impidiendo que ocurra cualquier otra cosa entre "Inicio" y "Fin".

- El **Ejemplo 4** soluciona lo anterior usando `Isolate.spawn`. Mueve la función pesada `sumaGrande` a un hilo separado, el `main` inicia el hilo y sigue ejecutando inmediatamente ("Mientras tanto, sigo ejecutando..."), demostrando que la interfaz o el hilo principal no se congelan mientras el otro hilo calcula.

- El **Ejemplo 3** ilustra la comunicación básica unidireccional. Se crea un `ReceivePort` en el `main` y se pasa su `SendPort` al Isolate nuevo. El Isolate envía un mensaje simple ("Hola...") y el `main` lo escucha.

- El **Ejemplo 5** implementa una comunicación bidireccional (ping-pong). El worker crea su propio puerto y se lo envía al `main`. Así, el `main` puede enviar mensajes

al worker y el worker puede responder. Se destaca el control del ciclo de vida con `mainReceive.close()` y `isolate.kill()` para finalizar los procesos limpiamente.

**Interpretación** Los conceptos que abarcan estos programas son Concurrencia, Paralelismo y Paso de Mensajes.

**Event Loop y Asincronía (Futures):** Dart es "single-threaded" por defecto. Los Ejemplos 1 y 2 demuestran cómo funciona el Event Loop. Usar `Future` no crea un nuevo hilo, sino que agenda una tarea para el futuro.

- Sin `await` (Ejemplo 1), el código sigue de largo (no bloqueante).
- Con `await` (Ejemplo 2), se simula un comportamiento síncrono, pausando la ejecución de esa función específica hasta que la tarea futura se completa, útil para esperar respuestas de servidores o temporizadores.

**Paralelismo Real (Isolates):** A diferencia de otros lenguajes que usan "Threads" compartiendo memoria, Dart usa Isolates.

- En el Ejemplo 6, vemos el concepto de bloqueo del hilo principal (CPU-bound blocking); si calculas algo pesado en el `main`, la aplicación se congela.
- En el Ejemplo 4, se aplica el paralelismo real. Al usar `Isolate.spawn`, se crea un nuevo espacio de memoria y un hilo de ejecución independiente. Esto permite que el cálculo pesado y la impresión de mensajes en el `main` ocurran simultáneamente.

**Memoria Aislada y Puertos:** El concepto clave de los Isolates es que no comparten memoria (no hay variables globales compartidas ni riesgo de *race conditions*).

- Para comunicarse, deben usar Paso de Mensajes a través de `SendPort` y `ReceivePort`, como se ve en los Ejemplos 3 y 5.

- Esto obliga a pensar en la arquitectura como un sistema de cliente-servidor, donde un hilo solicita algo y espera a que el otro le envíe la respuesta a través del puerto, en lugar de acceder a una variable común.

### 3.1.3. Patrones de Diseño

Este código implementa un sistema de gestión de impresiones centralizado utilizando el patrón de diseño Singleton para asegurar que solo exista una única instancia de la `Impresora` en toda la aplicación.

- **Clase de Datos (Documento):** Define la estructura básica de lo que se va a procesar, es una clase inmutable que contiene el id, usuario, nombre y contenido; sobrescribe el método `toString` para facilitar la visualización en consola.
- **Clase Singleton (Impresora):** Esta es la clase central del ejemplo.
  - **Instancia Única:** Declara una propiedad estática y final llamada `_instancia` que almacena la única referencia de la clase creada a través del constructor privado `_interna()`.
  - **Constructor Factory:** Utiliza la palabra clave `factory`. A diferencia de otros lenguajes donde se llama a un método `getInstance()`, en Dart el constructor `factory Impresora()` devuelve la instancia estática `_instancia` existente; esto permite usar `new Impresora()` (o solo `Impresora()`) pareciendo una instanciación normal, pero devolviendo siempre el mismo objeto.
  - **Gestión de Estado:** Maneja una `Queue` (Cola) llamada `_colaImpresion` para los documentos pendientes y una `List` para el historial, simulando el buffer de una impresora real.
- **Flujo del main:**
  - Se declaran dos variables, `impresoraA` e `impresoraB`, ambas instanciando `Impresora()`.

- Se demuestra la identidad con `identical(impresoraA, impresoraB)`, confirmando que ambas variables apuntan al mismo espacio de memoria.
- Se simula concurrencia de usuarios: `impresoraA` (Alice) e `impresoraB` (Bob) envían documentos y al imprimir con `impresoraA`, se procesan también los documentos enviados por `impresoraB`, demostrando que la cola es compartida.

**Interpretación** Los conceptos usados en este ejemplo son los siguientes:

**Patrón Singleton:** El objetivo es restringir la instanciación de una clase a un solo objeto. Una impresora física es un recurso compartido exclusivo; si cada usuario creara su propia instancia de "Impresora" en el software, habría múltiples colas de impresión desincronizadas enviando datos al hardware al mismo tiempo, lo cual causaría errores. El Singleton garantiza un punto de acceso global y una cola unificada.

**Constructores Factory:** Dart ofrece una forma elegante de implementar Singleton mediante constructores de fábrica. El cliente del código (`main`) no necesita saber que está usando un Singleton, simplemente llama a `Impresora()`; el constructor `factory` se encarga de la lógica de "si ya existe, devuélvelo; si no, créalo", aunque en el ejemplo la creación es inmediata al inicio.

**Estructura de Datos FIFO:** Para la lógica de impresión, se utiliza una Cola (`Queue`). Esto respeta el principio FIFO (*First In, First Out*): el primer documento que llega (`addLast`) es el primero en imprimirse (`removeFirst`); esto es esencial en sistemas de impresión para respetar el orden de llegada de los usuarios.

**Estado Compartido:** El ejemplo ilustra cómo el estado (`_colaImpresion` y `_historial`) persiste y es accesible desde diferentes "referencias"; aunque Alice (`impresoraA`) y Bob (`impresoraB`) parezcan tener impresoras distintas, ambos están modificando la misma lista en memoria.

## 3.2. Diagramas UML

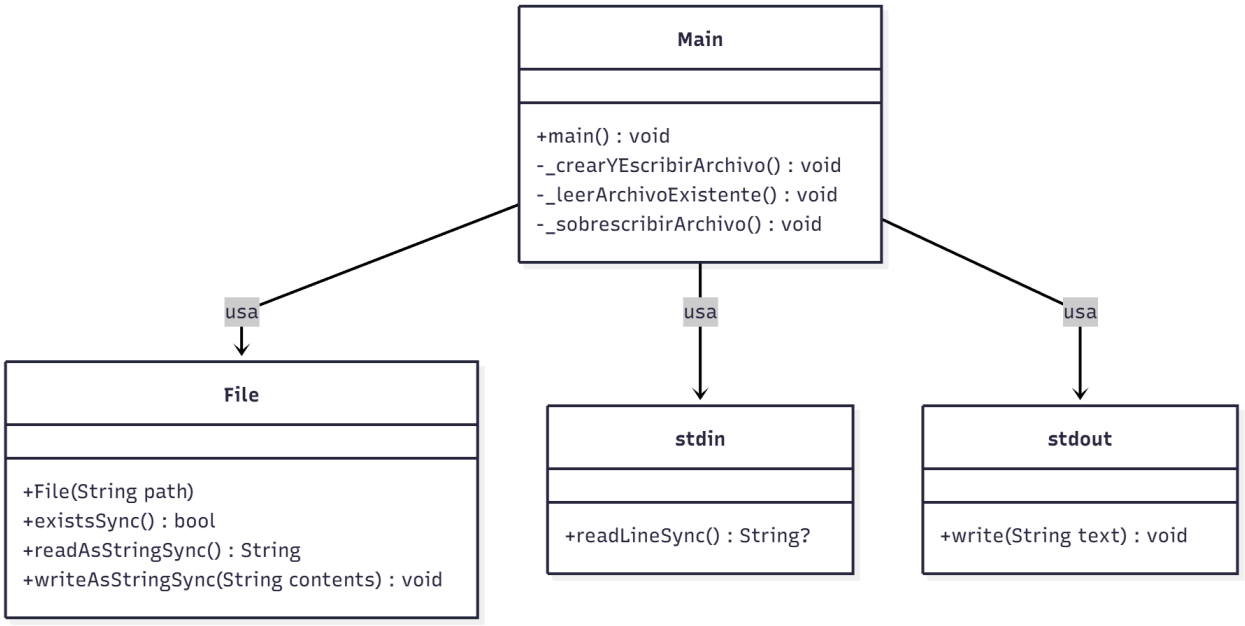


Figura 1: Archivos, Diagrama de Clases.

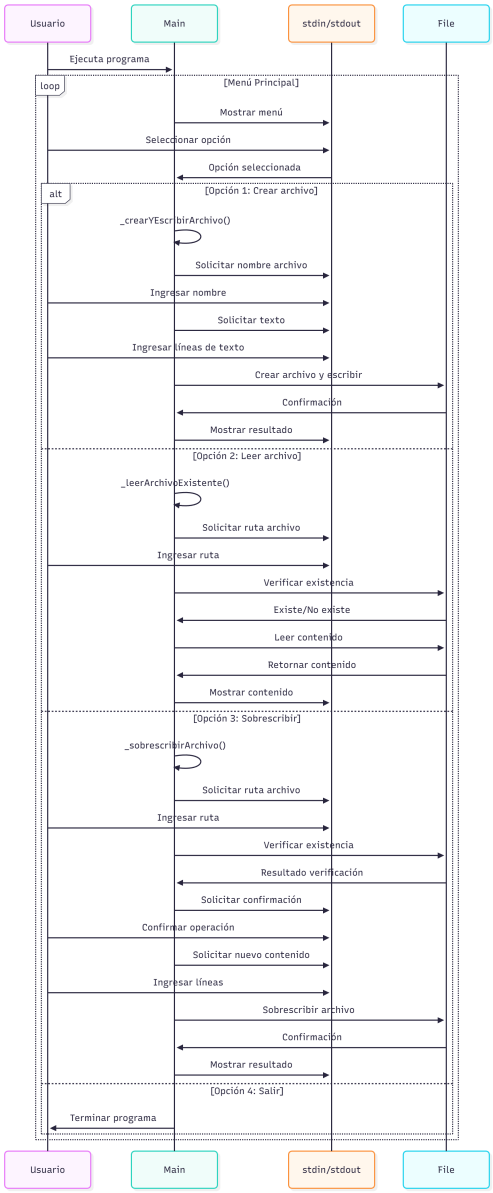


Figura 2: Archivos, Diagrama de Secuencia - Flujo Principal.

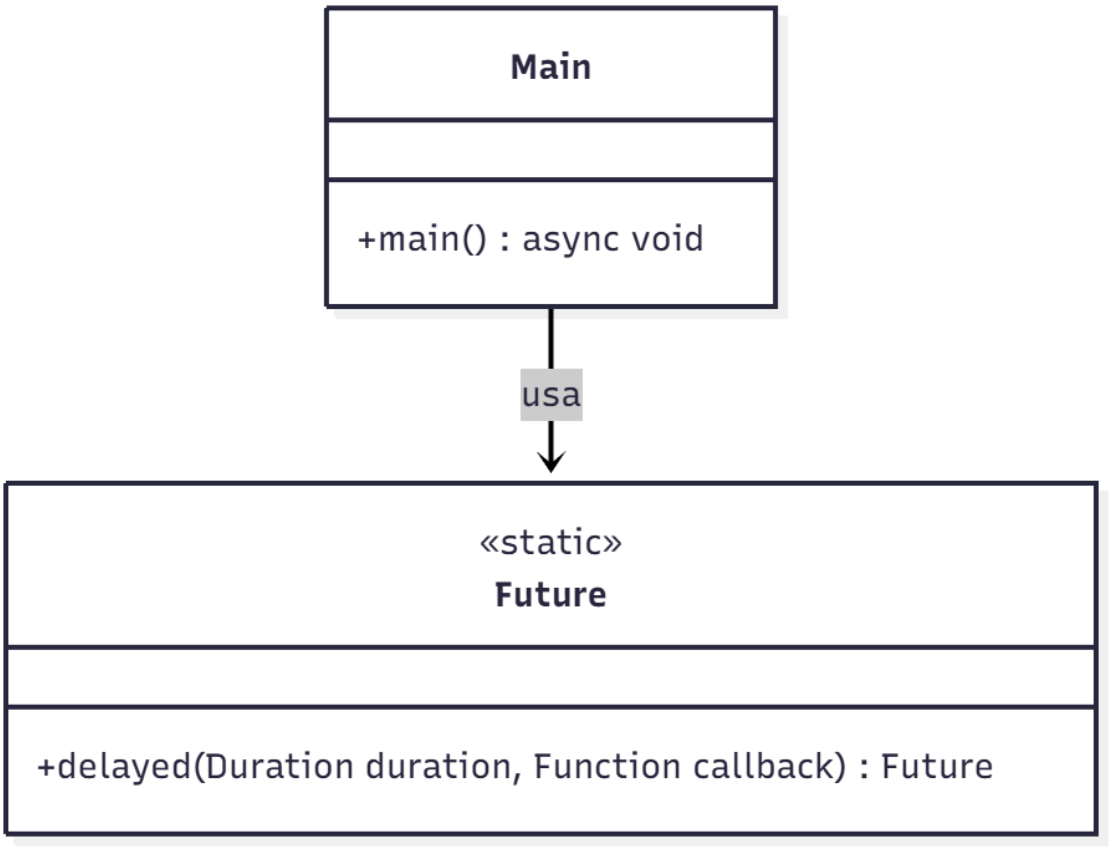


Figura 3: Hilos, Ejemplo 1, Diagrama de Clases.

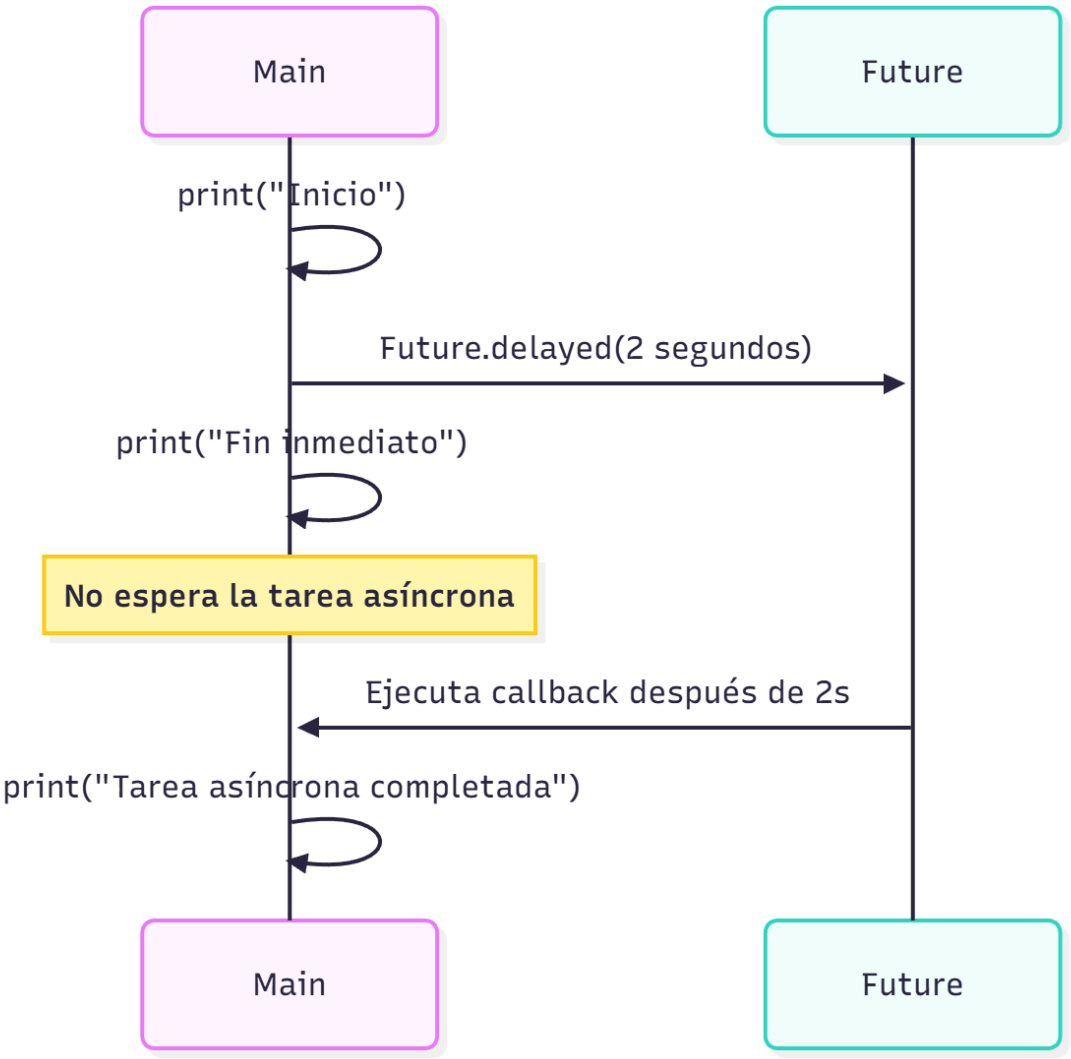


Figura 4: Hilos, Ejemplo 1, Diagrama de Secuencia.

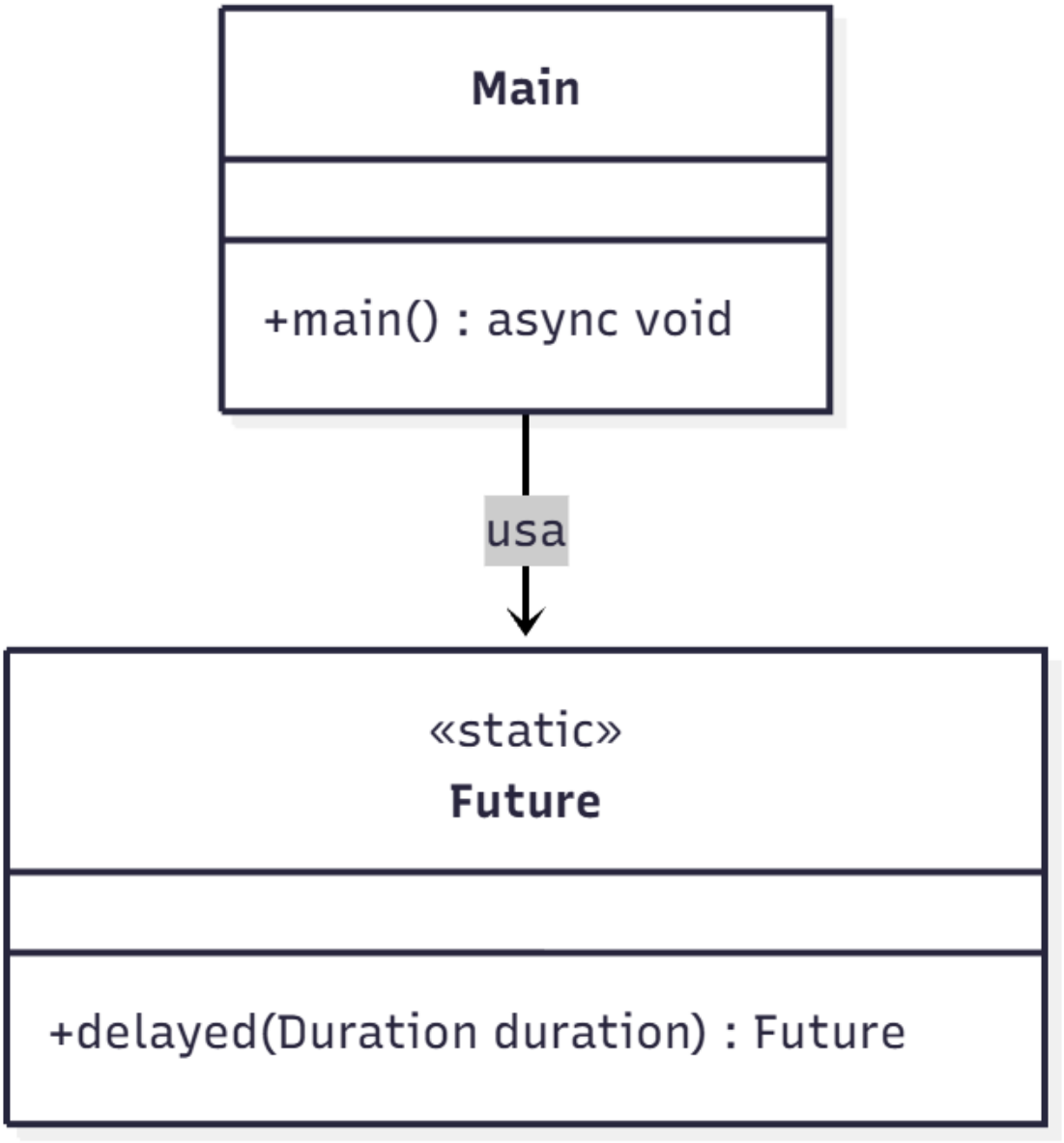


Figura 5: Hilos, Ejemplo 2, Diagrama de Clases.

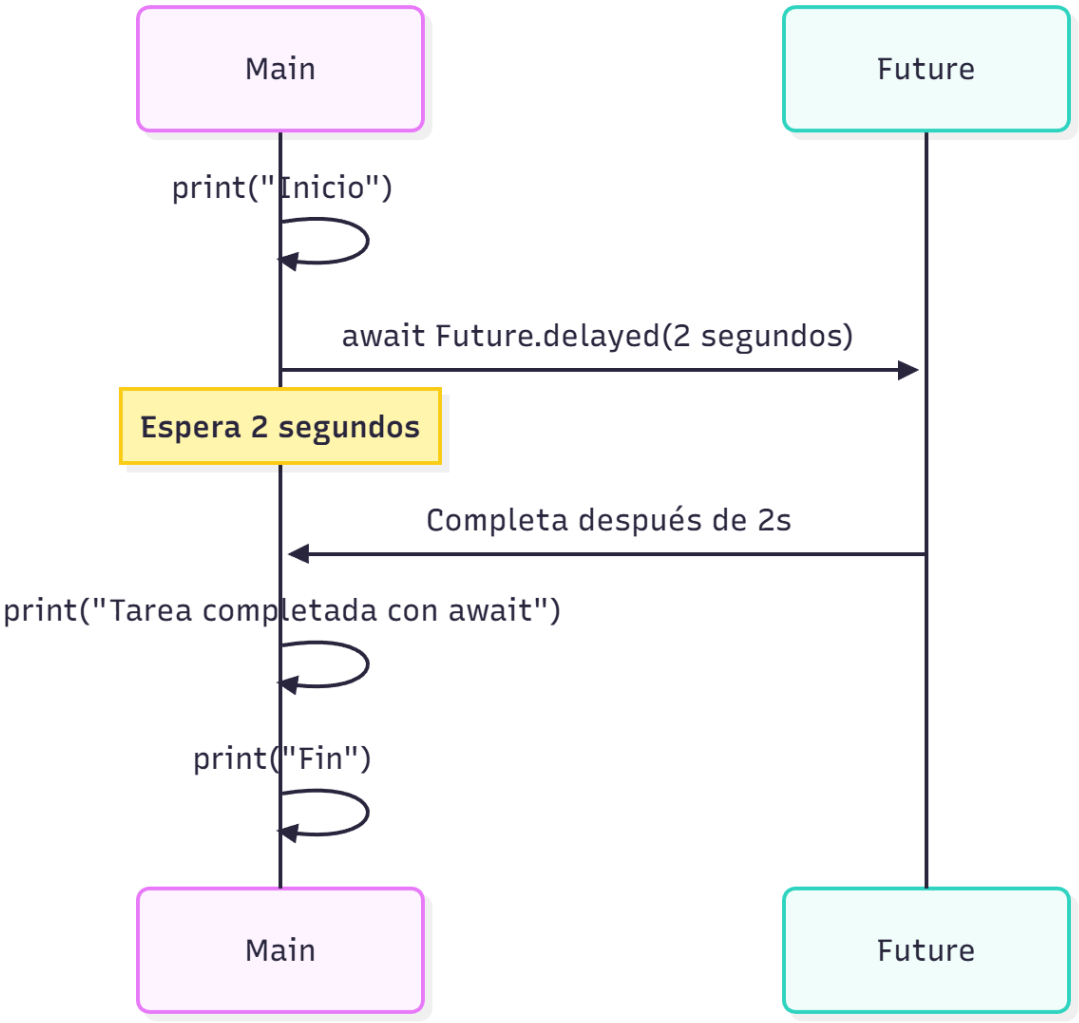


Figura 6: Hilos, Ejemplo 2, Diagrama de Secuencia.

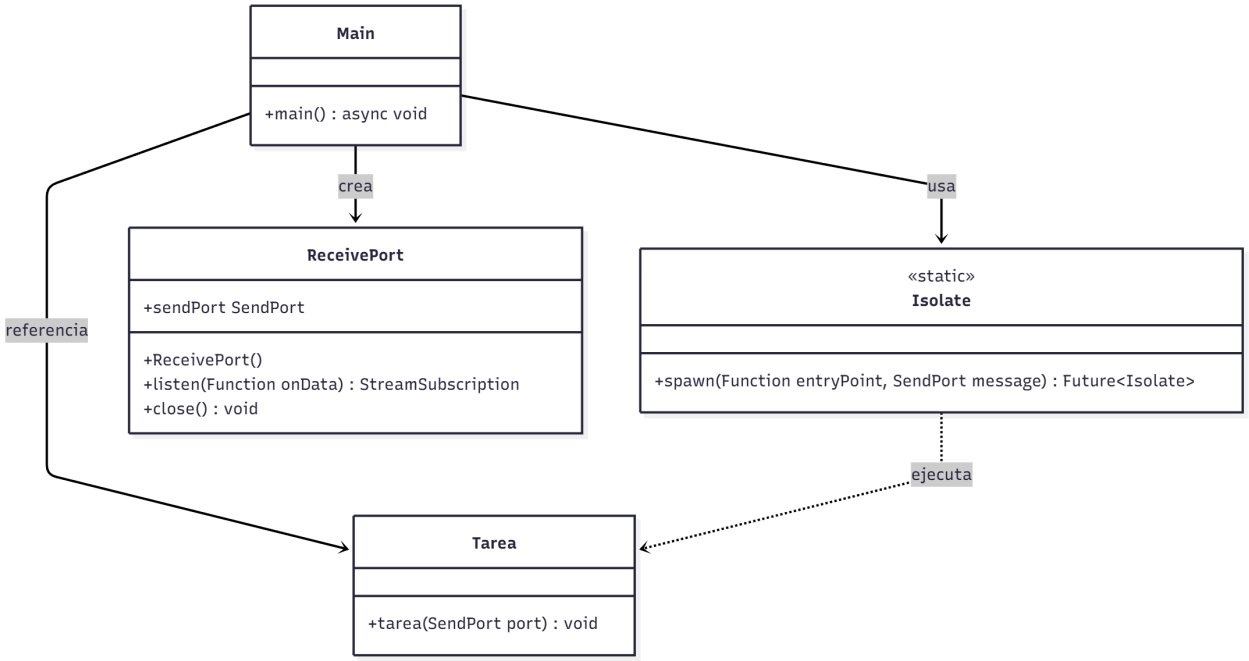


Figura 7: Hilos, Ejemplo 3, Diagrama de Clases.

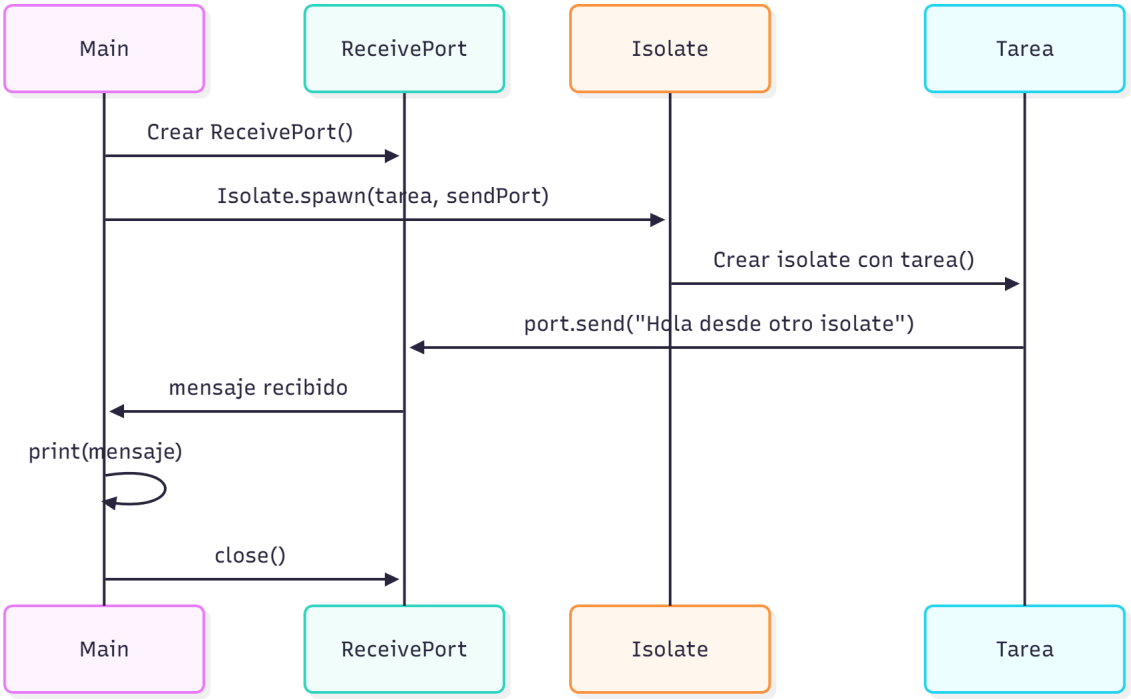


Figura 8: Hilos, Ejemplo 3, Diagrama de Secuencia.

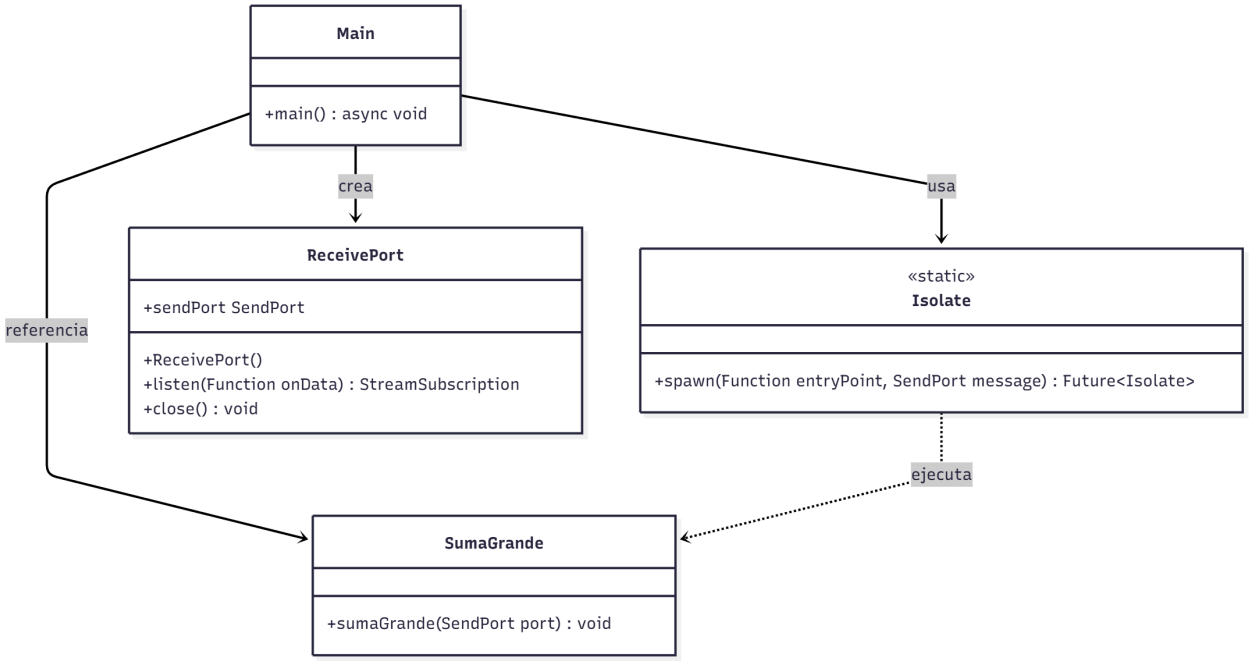


Figura 9: Hilos, Ejemplo 4, Diagrama de Clases.

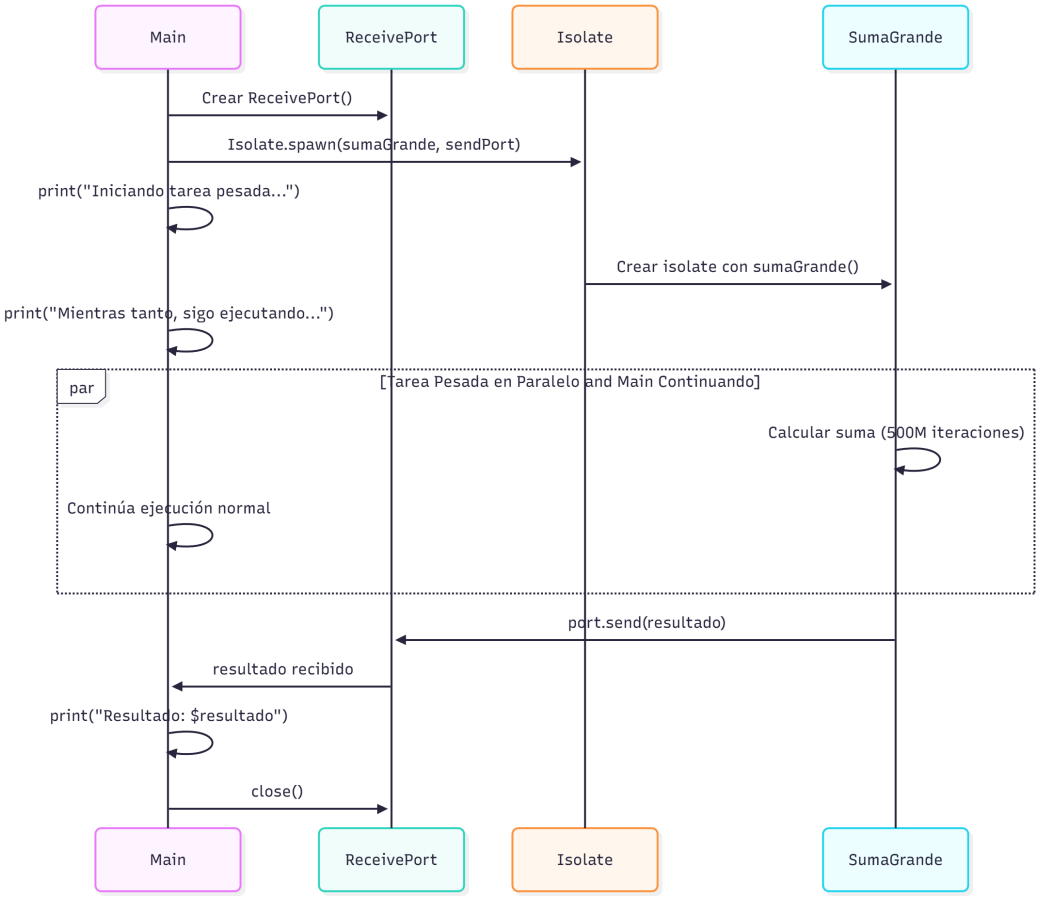


Figura 10: Hilos, Ejemplo 4, Diagrama de Secuencia.

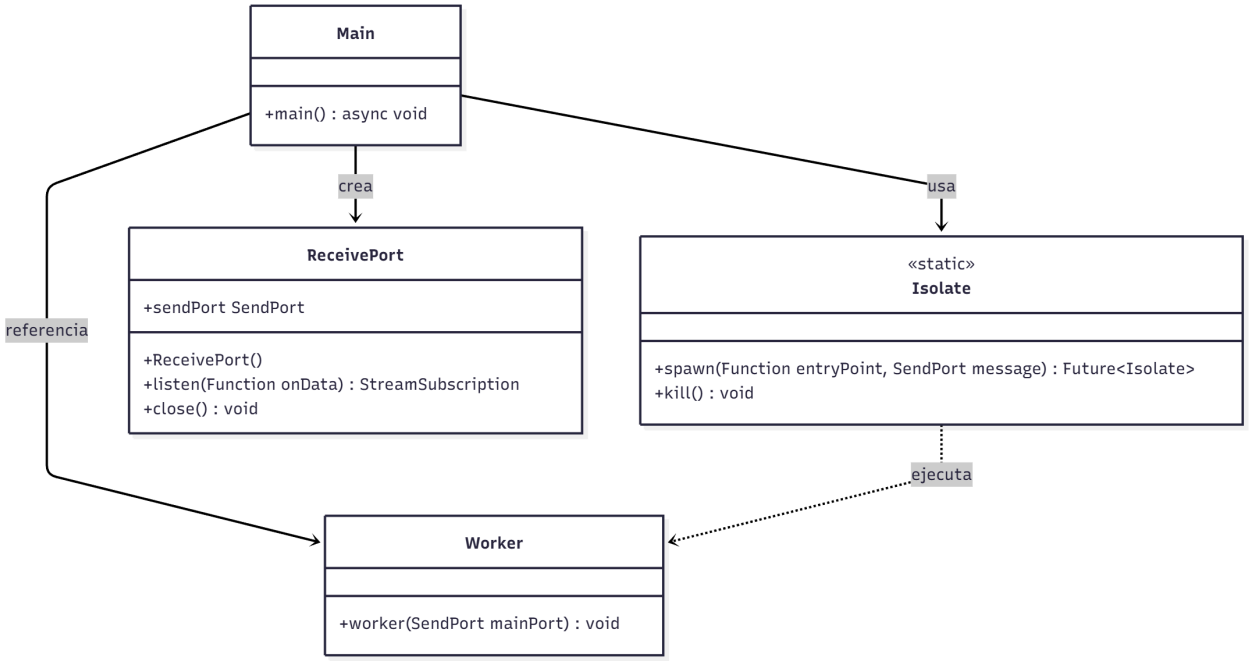


Figura 11: Hilos, Ejemplo 5, Diagrama de Clases.

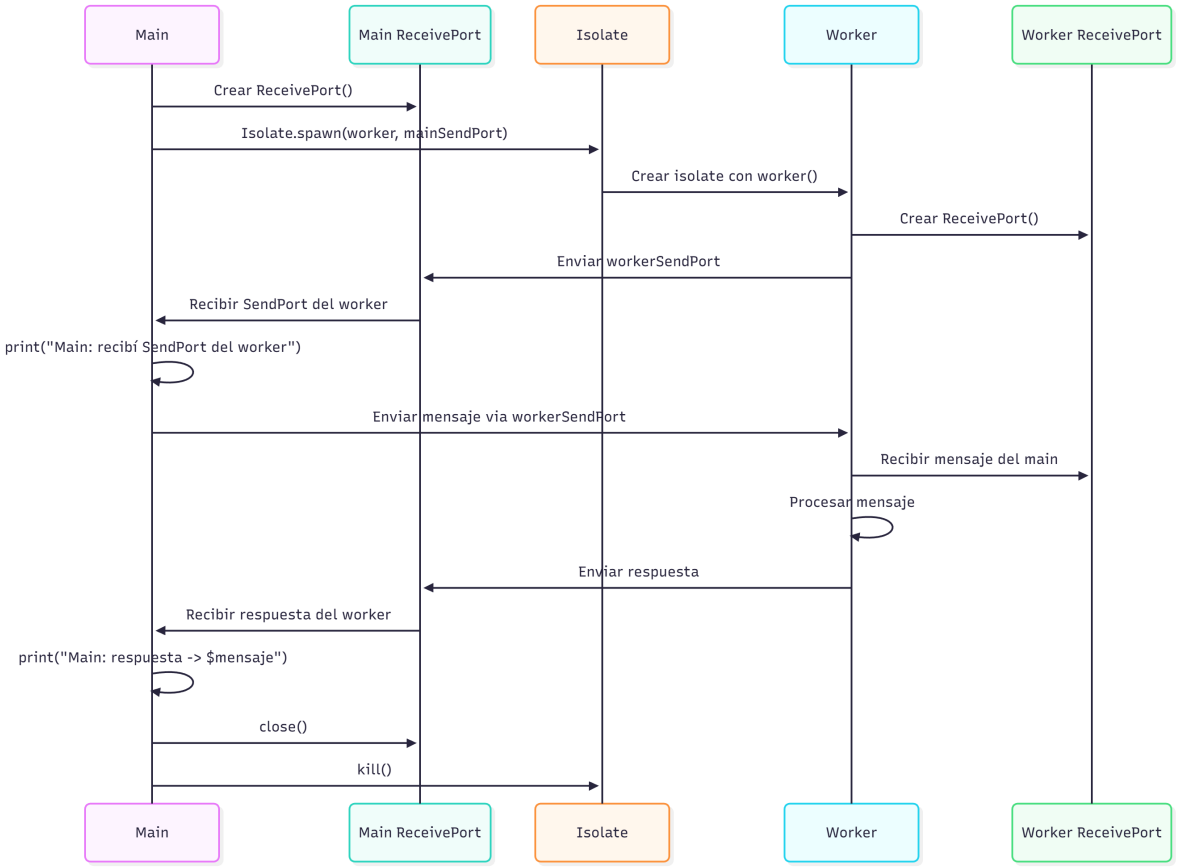


Figura 12: Hilos, Ejemplo 5, Diagrama de Secuencia.

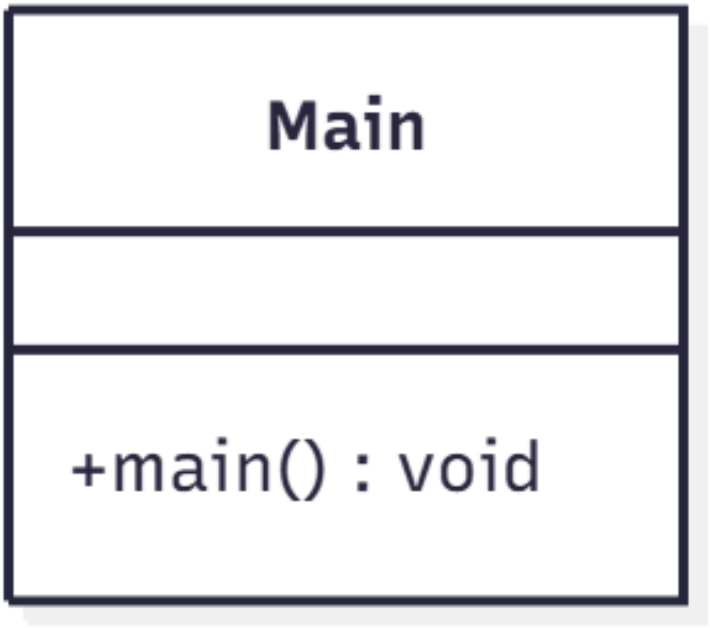


Figura 13: Hilos, Ejemplo 6, Diagrama de Clases.

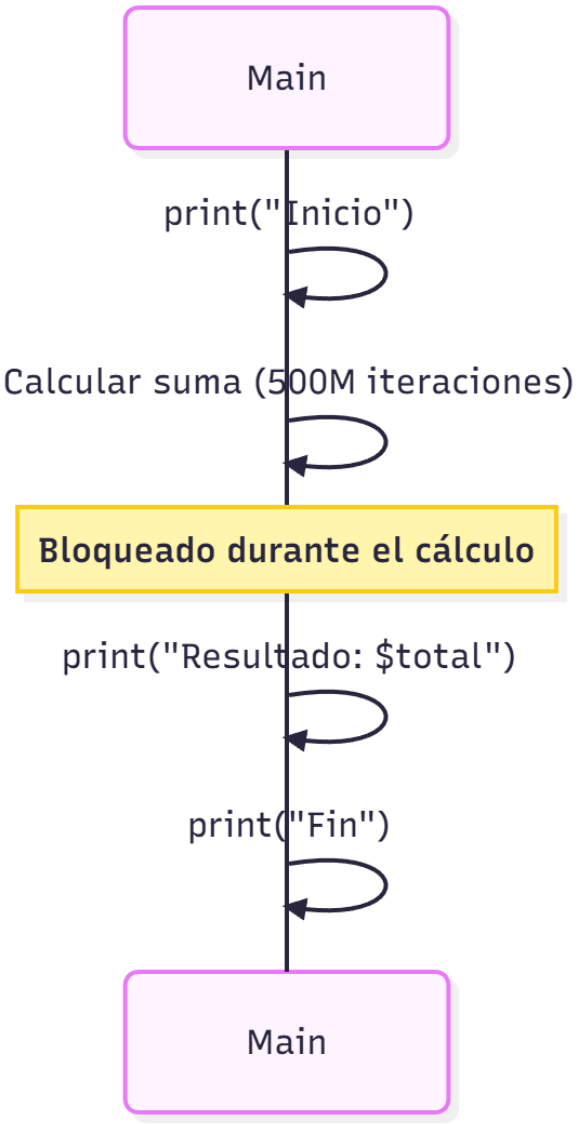


Figura 14: Hilos, Ejemplo 6, Diagrama de Secuencia.

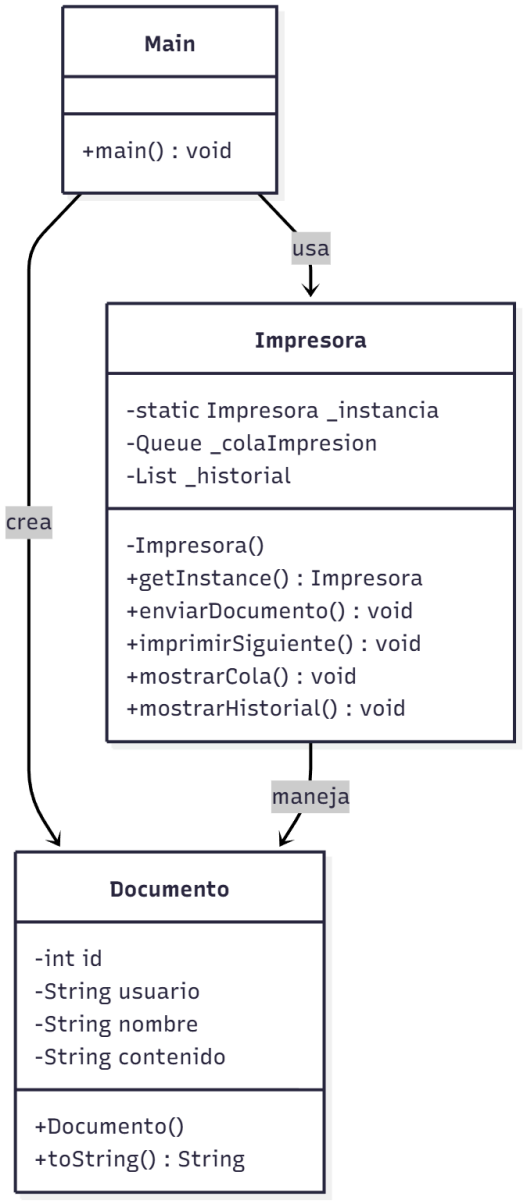


Figura 15: Patrón Singleton, Diagrama de Clases.

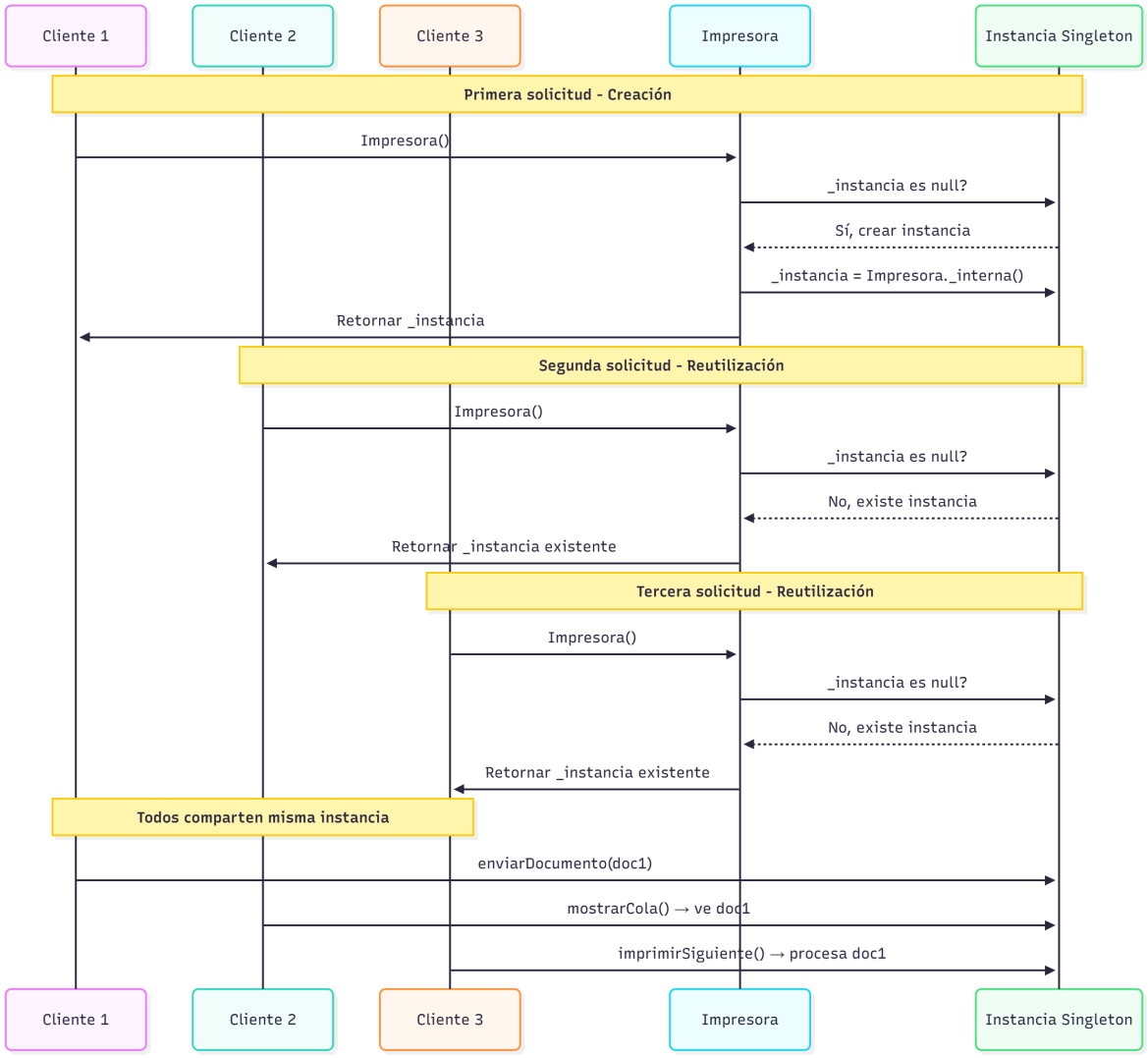


Figura 16: Patrón Singleton, Diagrama de Secuencia - Detalle de Singleton.

## 3.3. Archivos

### 3.3.1. Ejemplo 1: Crear archivo .txt y escribir texto

**Operación:** Creación de archivo con múltiples líneas **Archivo:** notas.txt **Contenido:**

```
Tareas pendientes:
Estudiar para el examen
Hacer la tarea de matemáticas
```

```
Leer el capítulo 4
```

- Se crea correctamente el archivo `notas.txt` en el directorio actual
- El contenido se guarda con saltos de línea entre cada línea ingresada
- El sistema confirma con mensaje: 'Archivo creado y guardado correctamente'
- Al leer el archivo, se muestra exactamente el texto ingresado

### 3.3.2. Ejemplo 2: Leer archivo existente

**Operación:** Lectura de archivo creado previamente **Archivo:** notas.txt

- El sistema busca el archivo en la ruta especificada
- Si existe, muestra el contenido completo con formato
- Se muestra el encabezado 'CONTENIDO DEL ARCHIVO' antes del contenido
- Si el archivo no existe, muestra mensaje: 'El archivo no existe'

### 3.3.3. Ejemplo 3: Sobrescribir archivo existente

**Operación:** Reemplazo completo de contenido **Archivo:** notas.txt **Nuevo contenido:**

```
Tareas actualizadas:
Reunión con el equipo
Presentación del proyecto
```

- El sistema verifica que el archivo existe antes de sobrescribir
- Solicita confirmación con 'SI' para proceder

- Elimina todo el contenido anterior y guarda el nuevo
- Confirma con mensaje: 'Archivo sobrescrito correctamente'

## 3.4. Hilos

### 3.4.1. Ejemplo 1: Future.delayed sin await

**Comportamiento:** Ejecución asíncrona no bloqueante

- Se imprime 'Inicio' inmediatamente
- Se programa tarea para 2 segundos después
- Se imprime 'Fin inmediato (sin esperar)' sin esperar
- Después de 2 segundos se ejecuta: 'Tarea asíncrona completada'
- El hilo principal no se bloquea durante la espera

### 3.4.2. Ejemplo 2: Future.delayed con await

**Comportamiento:** Ejecución secuencial con espera

- Se imprime 'Inicio' inmediatamente
- El programa espera 2 segundos completos
- Se imprime 'Tarea completada con await' después de la espera
- Finalmente se imprime 'Fin'
- El hilo principal se bloquea durante los 2 segundos

### 3.4.3. Ejemplo 3: Isolate básico

**Comportamiento:** Comunicación unidireccional entre isolates

- Se crea un ReceivePort en el isolate principal
- Se genera un nuevo isolate con la función `tarea`
- El isolate secundario envía mensaje: 'Hola desde otro isolate'
- El isolate principal recibe y muestra el mensaje
- Se cierra el puerto después de recibir el mensaje

### 3.4.4. Ejemplo 4: Isolate con cálculo pesado

**Comportamiento:** Procesamiento paralelo intensivo

- Se inicia cálculo pesado (suma de 500 millones) en isolate separado
- El hilo principal continúa ejecutándose inmediatamente
- Se imprime 'Mientras tanto, sigo ejecutando en el hilo principal...'
- El cálculo se ejecuta en paralelo sin bloquear la interfaz
- Cuando termina, se recibe el resultado: 'Resultado: 124999999750000000'

### 3.4.5. Ejemplo 5: Comunicación bidireccional entre isolates

**Comportamiento:** Diálogo completo entre isolates

- Isolate worker envía su SendPort al main
- Main recibe confirmación: 'Main: recibí el SendPort del worker'
- Main envía mensaje: "Mensaje desde main"
- Worker responde: 'Recibido en worker: Mensaje desde main'

- Main recibe respuesta y cierra la comunicación
- Se mata el isolate worker para liberar recursos

#### 3.4.6. Ejemplo 6: Cálculo sincrónico (Sin hilos)

**Comportamiento:** Procesamiento bloqueante

- Se imprime 'Inicio'
- El programa se bloquea completamente durante el cálculo
- No responde a entradas del usuario durante 5-10 segundos
- Al terminar, muestra: 'Resultado: 124999999750000000'
- Finalmente imprime 'Fin'
- Demuestra la necesidad de hilos para operaciones intensivas

### 3.5. Patrón Singleton

#### 3.5.1. Ejemplo: Sistema de Impresión con Singleton

**Patrón:** Singleton **Documentos enviados:**

- Documento 1: 'Reporte mensual' (Usuario: Alice)
- Documento 2: 'Contrato de servicio' (Usuario: Bob)
- Documento 3: 'Presentación proyecto X' (Usuario: Carlos)
- Verificación Singleton: `identical(impresoraA, impresoraB)` retorna `true`
- Todos los documentos se agregan a la misma cola compartida
- La cola muestra 3 documentos en orden FIFO

- Al imprimir, se procesan en el orden de llegada
- El historial acumula todos los documentos impresos
- La cola se vacía después de procesar todos los documentos
- El estado se mantiene consistente entre todas las referencias

**Resultados esperados:**

- Cola inicial: 3 documentos pendientes
- Después de 2 impresiones: 1 documento pendiente
- Historial final: 3 documentos impresos
- Estado consistente en todas las referencias a la impresora

# 4. Conclusiones

El estudio de Archivos, Hilos de Ejecución y Patrones ha culminado en la adquisición exitosa de las competencias teóricas y prácticas necesarias para el desarrollo de software profesional. Se logró un entendimiento profundo del manejo del flujo de datos y la persistencia; la gestión eficiente de tareas concurrentes; y la aplicación de soluciones estructurales como el patrón Singleton. Este conocimiento se consolidó eficazmente gracias a la ejecución y análisis detallado de los códigos y ejemplos proporcionados por el profesor, lo cual permitió visualizar la aplicación de cada concepto en un entorno práctico real.

# Referencias


[1] J.L. Blasco. *Introducción a POO en Java: Excepciones*. URL: https://openwebinars.net/blog/introduccion-a-poo-en-java-excepciones/.

[2] Virginia Calpena. *Excepciones*. 2016. URL: //vcalpena.wordpress.com/4-excepciones/.

[3] Paul Deitel y Harvey Deitel. *Java: How to Program, Early Objects (11a Edición)*. Pearson. 2017.

[4] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional. 1994.

[5] Oracle. *The Java Tutorials: Concurrency*. 2023. URL: https://docs.oracle.com/javase/tutorial/essential/concurrency/.

[6] Dionisio Pérez Pérez. *Flujos y Archivos*. 2012. URL: https://pooitsavlerdo.blogspot.com/2012/06/6-flujos-y-archivos.html.

[7] Unknown. *Tipos de excepciones*. URL: https://itzoisc.blogspot.com/2012/06/52-tipos-de-excepciones.html.