

Practica-910-Desarrollo

DANIEL S ANGUIANO R

November 2025

1. Código del tema Archivos

La aplicación desarrollada permite la creación, lectura y sobreescritura de archivos tipo .txt implementando los conceptos de manejo de errores y archivos, pues el programa implementa **dart.io** para la entrada/ salida de datos y basa su funcionamiento en el flujo de datos que se produce entre el porograma y la fuente. Explicando más a fondo la aplicación:

1.1. main()

Es la clase principal en la que se imprime un menú con las e opciones disponibles para trabajar con archivos de texto. Cada opción a excepción de la de salir en la que solo se muestra un mensaje, llama un función de tipo privada para realizar la operación en cuestión, ofreciendo así un código más organizado.

1.2. Funciones:

1.2.1. crearYEscribirArchivo()

La función tiene el propósito de crear un archivo de tipo .txt y permitirle al usuario ingresar el contenido, donde el contenido admitido solo son Strings. Una vez que se confirma que el nombre del archvio no es NULL o vacío, utiliza una lista de Strings para agregar las lineas escritas y termina una vez que detecta que se escribió FIN.

Las líneas escritas en la lista son agregadas a un archivo **FILE** donde se implementa manejo de errores al poner el bloque de código dentro de un try/catch para cubrir el caso en el que ocurra un error al guardar el archivo.

1.2.2. leerArchivoExistente()

La función tiene el propósito de leer el contenido de un archivo que se busca a través de una ruta proporcionada por el usuario, una vez que se comprueba que la ruta es válida se implementa un try para comprobar la existencia del archivo con

`existsSync()`. Si el archivo existe, el contenido se lee como cadena y se imprime en consola.

1.2.3. sobrescribirArchvio()

La función tiene el propósito de buscar un archivo a través de una ruta proporcionada por el usuario, donde se verifica la validez de la ruta y la existencia del archivo que se está buscando. Se lanza un mensaje de confirmación para la sobreescritura del archivo, en caso de ingresar **SI** se continua con la sobreescritura, guardando las nuevas líneas de texto en la lista `nuevasLineas []`, cuando se detecte la palabra **FIN** se implementa un `try/catch` para ingresar el nuevo texto al archivo, mandando un mensaje de error en caso de que falle la operación,

2. Ejemplos del tema Hilos

2.1. Ejemplo 1:

El primer ejemplo es de concurrencia básica donde se utilizan **Future** para marcar una tarea que se completará a futuro sin interferir con el hilo principal y **async** para marcar que la tarea es asíncrona. Así la tarea Inicio se completa y espera dos segundos a que la tarea `Future.deployed` terminé su ejecución, imprimiendo los mensajes en consola como si hubiera sido una sola tarea que se ejecutó en secuencia.

2.2. Ejemplo 2:

Se usa **Future** y **async** para señalar que la función es asíncrona y que puede ejecutar tareas que requieran tiempo de espera además de la opción de utilizar **await** trabajando con la concurrencia. Así funciona imprimiendo un mensaje inicial, pero el **await** señala que el programa podrá continuar una vez que espere el resultado de `Future.deployed`, finalizando en un tiempo de 2 segundos.

2.3. Ejemplo 3:

El tercer ejemplo muestra un caso básico de uso de **Isolate** usando la comunicación por mensajes. Así se crea una función `tarea` donde se define el método `SendPort` utilizado para mandar mensajes al isolate principal. Por otro lado el isolate principal define el método escucha `receivePort` para recibir el mensaje y usa **async** para usar **await** haciendo que el programa espere a que se cree un nuevo isolate donde se ejecute el método `tarea` indicando con `receivePort.sendPort` lo que va a recibir, extrayendo el mensaje con el método escucha `receivePort.listen()` y por último imprimiendo el mensaje en pantalla y cerrando el `receivePort` ya que no hay necesidad de recibir mensajes.

2.4. Ejemplo 4:

El cuarto ejemplo consiste en realizar la suma de cero a cinco millones sin bloquear el hilo principal, implementando los isolates para hacer la operación pesada de manera independiente sin interrumpir el flujo del isolate principal. Así se crea la función `sumaGrande()` que define su método `sendPort`. Por otra parte el isolate principal define su método escucha `receive` y usa `await` para seguir el flujo una vez que se cree el isolate paralelo y ejecute la operación mientras el flujo del isolate principal se prepara para recibir el resultado, imprimiéndolo una vez que el hilo paralelo acaba la sumatoria.

2.5. Ejemplo 5:

En este ejemplo se implementa una comunicación bidireccional entre dos isolates, uno se encarga de mandar y otro de recibir, para lo cual ambos deben definir sus métodos `receivePort` y `sendPort`. Así la función `worker` define sus métodos y a través del método escucha `receive.listen()` recibe los mensajes del `main` y manda una respuesta con `mainPort.send()`. Por otra parte el `main()` crea un isolate donde se ejecuta `worker` y así recibe su `sendPort()` creando su método escucha y definiendo un `if/else` para confirmar cuando se establezca la comunicación obteniendo como primer mensaje el `sendPort` y como segundo mensaje la respuesta de `worker`, matando el isolate al finalizar para devolver el control al sistema.

2.6. Ejemplo 6:

Este último ejemplo muestra el caso de ejecución del ejemplo 4 donde se realiza la sumatoria, pero ahora realizando todo en un solo hilo mostrando que al final el tiempo de ejecución es despreciable, pero recordando que la utilidad de los hilos está en la eficiencia de la ejecución y la optimización de recursos.

3. Patrones de diseño

La implementación se estructuró siguiendo el patrón de arquitectura Modelo-VistaControlador, organizando el código en tres capas principales.

3.1. Capa de modelo (Clases Pokemon y Ataque)

La clase `Pokemon` modela las características base de los pokémones, posee atributos finales como nombre, nivel y tipo, además de atributos calculados dinámicamente como vida y velocidad. En el constructor, se utiliza la clase `Random` para asignar valores aleatorios a estas estadísticas basándose en el nivel del pokémon. Se implementaron las subclases `PokemonFuego` y `PokemonHierba`, las cuales heredan de `Pokemon` e inicializan el tipo específico mediante la llamada al constructor de la superclase con el método `super`.

De manera análoga, la clase Ataque define la estructura de los movimientos con atributos para nombre, tipo y potencia. Se extendió esta funcionalidad a través de las clases AtaqueFuego, AtaqueHierba y AtaqueNormal, permitiendo clasificar las acciones para el combate.

3.2. Capa de vista (Clases CombateView)

Se definió una clase abstracta CombateView, que establece los métodos necesarios para la interacción con el usuario, tales como mostrarInformacionPokemon, mostrarDanio o mostrarGanador. La implementación concreta se realizó en la clase ConsoleCombateView, la cual define el comportamiento de estos métodos utilizando la función print para desplegar los eventos y resultados del combate directamente en la consola.

3.3. Capa de controlador (Clase CombateController)

La clase CombateController es quien gestiona el flujo del programa. Cuenta con un atributo de tipo CombateView para comunicarse con la interfaz. Su lógica central reside en el método iniciarCombate, el cual recibe a los pokemones y gestiona el combate dentro de un ciclo while que se mantiene activo mientras la vida de ambos pokemones sea mayor a cero.

Dentro de este ciclo, se determina el orden de los turnos comparando el atributo velocidad de cada pokemon, en caso de empate, se utiliza Random para decidir aleatoriamente quién ataca primero. El cálculo del daño se realiza con el método privado `_atacar`, donde se evalúa la efectividad de tipos verificando si el tipo del ataque tiene ventaja contra el tipo del defensor, asignando un multiplicador de 2.0 para ataques "súper efectivos." o de 0.5 para "poco efectivos". Finalmente, se actualiza el estado del defensor y se notifica a la vista.

3.4. Método principal (Main)

En la función main se realiza la instanciación e inicialización de los componentes. Se crea primero la vista ConsoleCombateView y se usa en el constructor del CombateController. Posteriormente, se instancian los objetos de tipo PokemonFuego y PokemonHierba con sus nombres y niveles, así como un objeto de tipo Ataque. Finalmente, se invoca el método iniciarCombate del controlador, pasando los objetos creados como argumentos para dar inicio a la ejecución del programa.