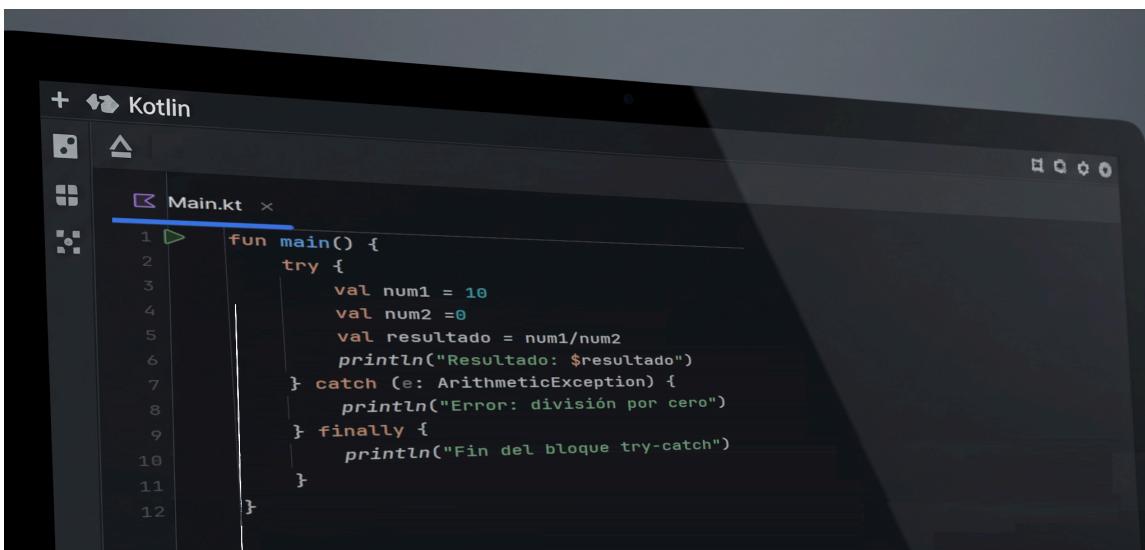


# UD1 - Manejo de ficheros



A screenshot of a dark-themed code editor showing a Kotlin file named Main.kt. The code contains a main() function with a try-catch block. It attempts to divide num1 by num2, prints the result, and catches an ArithmeticException if num2 is zero, printing an error message. Finally, it prints a message indicating the end of the try-catch block.

```
fun main() {
    try {
        val num1 = 10
        val num2 = 0
        val resultado = num1/num2
        println("Resultado: $resultado")
    } catch (e: ArithmeticException) {
        println("Error: división por cero")
    } finally {
        println("Fin del bloque try-catch")
    }
}
```

Imagen creada con ayuda de Gemini

## Resumen

En este documento se recogen los contenidos referentes al RA1 (desarrolla aplicaciones que gestionan información almacenada en ficheros identificando el campo de aplicación de los mismos y utilizando clases específicas).

## Revisores

Revisión	Fecha	Descripción
1.0	21-08-2025	Gestión de información almacenada en ficheros
2.0	17-09-2025	Numerar ejemplos y añadir prácticas
3.0	24-09-2025	Renombrar y ampliar apartado 6
4.0	26-09-2025	Modificaciones apartado 7
5.0	02-10-2025	Reestructurar algunas prácticas



Obra realizada por Begoña Paterna Lluch basada en materiales desarrollados por Alicia Salvador Contreras. Publicada bajo licencia [Creative Commons Atribución/Reconocimiento-CompartirIgual 4.0 Internacional](#).

## ÍNDICE

1. Introducción	1
🎯 Práctica 1: Proyecto Kotlin con Gradle	3
2. Gestión de ficheros y directorios	4
🎯 Práctica 2: Directorios y comprobaciones	10
3. Ficheros de texto	10
4. Ficheros de intercambio de información	12
4.1. CSV (Comma-Separated Values)	15
🎯 Práctica 3: Creación de tu CSV	19
4.2. XML (eXtensible Markup Language)	19
🎯 Práctica 4: Creación de tu XML	23
4.3. JSON (JavaScript Object Notation)	23
🎯 Práctica 5: Creación de tu JSON	27
4.4. Conversiones entre ficheros	27
🎯 Práctica 6: Elige tu fichero de datos	28
📁 Entrega parcial	28
5. Ficheros binarios	28
5.1. Ficheros binarios no estructurados	29
5.2. Ficheros binarios estructurados	30
5.3. Ficheros de imágenes	32
6. Ficheros binarios de acceso aleatorio	35
🎯 Práctica 7: Gestión de ficheros binarios	46
7. Documentación: El Fichero LEEME.md	47
🎯 Práctica 8: El Producto Final	49
📁 Entrega final	49
ANEXO 1: Guía de calificación	50

## Guía de uso

Estos apuntes están diseñados para que aprendas haciendo. A lo largo de la unidad, no solo veremos la teoría, sino que la aplicaremos directamente para construir, paso a paso, una aplicación completa de gestión de datos. El tema de la aplicación lo eliges tú, pero los pasos que daremos serán los mismos para todos.

Siguiendo la unidad no solo habrás aprendido los conceptos, sino que tendrás una aplicación completa y funcional creada por ti.

Intercaladas con la teoría y con los ejemplos encontrarás tres tipos de cajas:

-  Ejecutar y analizar
-  Práctica para aplicar y construir
-  Entrega

A continuación se explica que es cada una de ellas:

### Ejecutar y analizar

Estas cajas son para **analizar y comprender** en detalle el ejemplo de código proporcionado. Tu tarea es ejecutar ese código, observar la salida y asegurarte de entender cómo y por qué funciona.

### Práctica: Aplicar y construir

Estas cajas son **prácticas que debes realizar tú**. Es el momento de ponerte a programar y aplicar lo que acabas de aprender. Son los objetivos que debes completar para avanzar.

Cada una de estas prácticas es un bloque que debes programar para ir avanzando en tu proyecto final. En cada práctica ampliarás lo de las anteriores.

### Entrega

Estas cajas son **entregas de tu trabajo**. Las entregas pueden ser **parciales** (la profesora te dará sugerencias de mejora) o **finales** (la profesora calificará el trabajo que has realizado). No todas las prácticas llevan asociada una entrega).

## 1. Introducción

Un **fichero o archivo** es una unidad de almacenamiento de datos en un sistema informático. Es un conjunto de información (secuencia de bytes) organizada y almacenada en un dispositivo de almacenamiento (disco duro, memoria USB o un servidor en la nube). Los datos guardados en ficheros persisten más allá de la ejecución de la aplicación que los trata. La utilización de ficheros es una alternativa sencilla y eficiente a las bases de datos.

Características de un fichero:

- **Nombre:** Cada fichero tiene un nombre único dentro de su directorio.
- **Extensión:** Indica su tipo (.txt para texto, jpg para imágenes, etc).
- **Ubicación:** Directorios (carpetas) dentro del sistema de ficheros.
- **Contenido:** Texto, imágenes, vídeos, código fuente, bases de datos, etc.
- **Permisos de acceso:** Se pueden configurar para permitir o restringir la lectura, escritura o ejecución a determinados usuarios o programas.

Tipos de ficheros:

- **De texto:** Formato legible por humanos (.txt, .csv, .json, .xml).
- **Binarios:** Formato no legible directamente (.exe, jpg, .mp3, .dat).
- **De código fuente:** Contienen instrucciones escritas en lenguajes de programación (.java, .kt, .py).
- **De configuración:** Almacenan parámetros de configuración de programas (.ini, .conf, .properties, .json).
- **De bases de datos:** Se utilizan para almacenar grandes volúmenes de datos estructurados (.db, .sql).
- **Historial:** de eventos o errores en un sistema (.log).

API para manejo de ficheros:

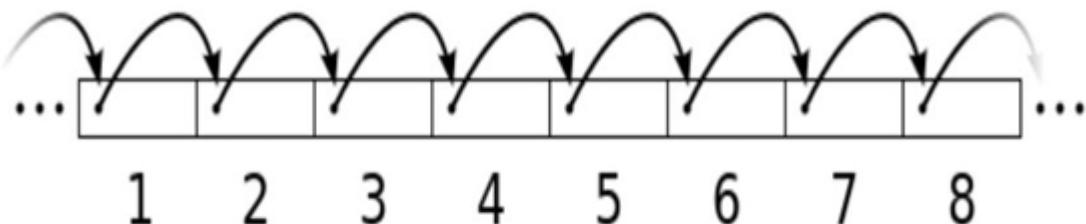
**Java.nio** (New IO) es una API disponible desde la versión 7 de Java que permite mejorar el rendimiento, así como simplificar el manejo de muchas operaciones. Funciona a través de interfaces y clases para que la máquina virtual Java tenga acceso a ficheros, atributos de ficheros y sistemas de ficheros. En los siguientes apartados veremos cómo trabajar con ella.

## Formas de acceso:

El acceso a ficheros es una tarea fundamental en la programación, ya que permite leer y escribir datos persistentes. Hemos visto que hay diferentes tipos de ficheros, según sus características y necesidades existen dos formas principales de acceder a un fichero (secuencial y aleatorio):

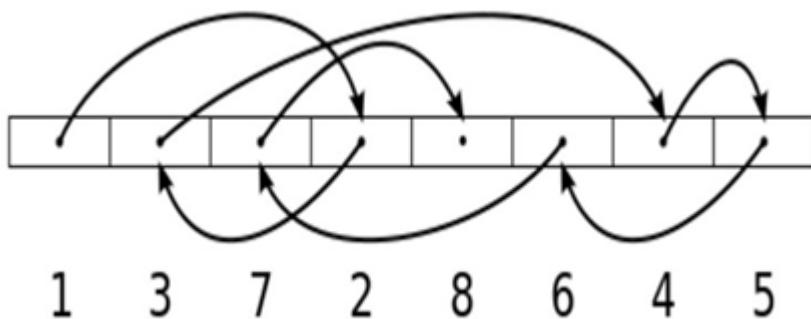
### Acceso secuencial

- Los datos se procesan en orden, desde el principio hasta el final del fichero.
- Es el más común y sencillo.
- Se usa cuando se desea leer todo el contenido o recorrer registro por registro. Por ejemplo lectura de un fichero de texto línea por línea, o de un fichero binario estructurado registro a registro.



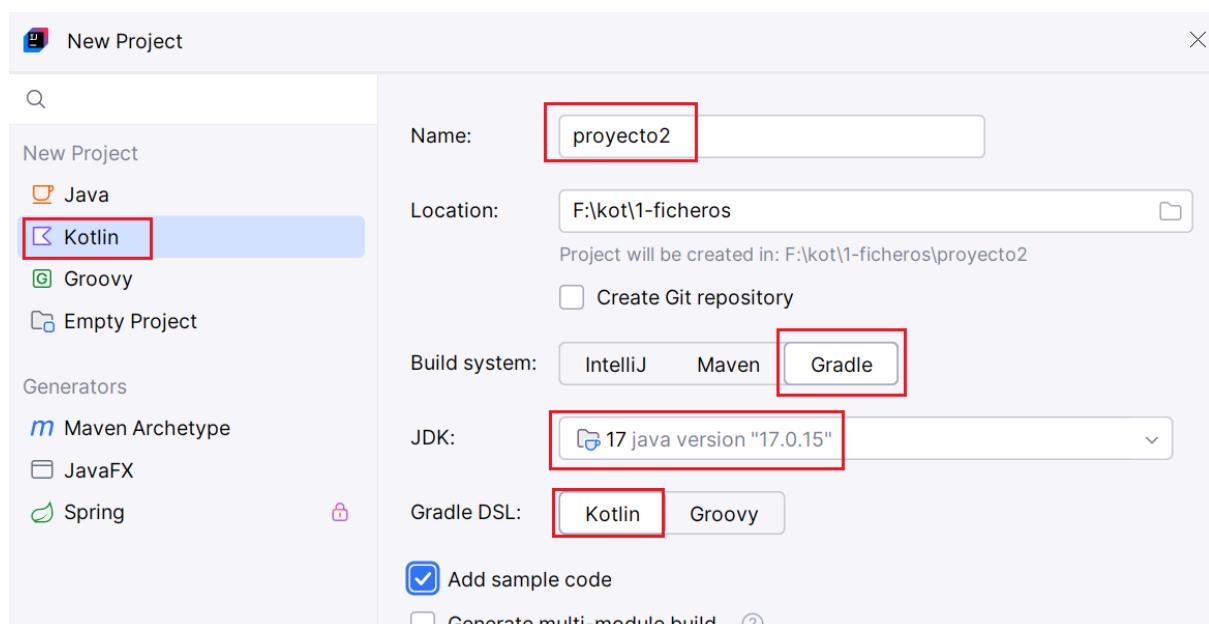
### Acceso aleatorio

- Permite saltar a una posición concreta del fichero sin necesidad de leer lo anterior.
- Es útil cuando los registros tienen un tamaño fijo y se necesita eficiencia (por ejemplo, ir directamente al registro 100).
- Requiere técnicas más avanzadas como el uso de FileChannel, SeekableByteChannel o RandomAccessFile.



A lo largo de esta unidad se explicarán algunas funciones de manejo de ficheros que requieren librerías externas (dependencias). Utilizaremos **Gradle** para descargarlas automáticamente en nuestros proyectos.

Para crear un proyecto Kotlin con Gradle en IntelliJ haremos clic en *New Project*, indicamos la información de la siguiente imagen, haremos clic en el botón *Create* y esperaremos a que IntelliJ prepare el proyecto.



A medida que necesitemos utilizar dependencias en nuestro proyecto, las iremos añadiendo al fichero **build.gradle.kts** en la sección de dependencias. Si después de añadirlas no se descargan automáticamente, abrir la **ventana Gradle** (lateral derecho de IntelliJ) y hacer clic en el botón de actualizar.

### 🎯 Práctica 1: Proyecto Kotlin con Gradle

En esta práctica has de crear un proyecto que irás ampliando a lo largo de toda la unidad. Realiza lo siguiente:

1. **Piensa** en una aplicación de gestión orientada al sector que prefieras y busca un nombre original (será el nombre de tu proyecto).
2. **Crea** un nuevo proyecto con Gradle y comprobar que se ejecuta correctamente (puedes utilizar el código de ejemplo de IntelliJ).

## 2. Gestión de ficheros y directorios

La gestión de ficheros y directorios se realiza a través de **Path** y **Files**.

**Path:** Representa una **ruta** en el sistema de ficheros (ej. /home/usuario/foto.png o C:\usuarios\docs\informe.txt). Un objeto Path es una dirección y no significa que el fichero o directorio exista. Algunos de sus principales métodos son:

Método	Descripción
Path.of(String)	Crea un objeto Path a partir de un String de ruta (Java 11+). Por debajo llama a Paths.get() que es el método original de la clase Paths (Java 7+)
toString()	Devuelve la ruta como un String (se llama por defecto desde <i>println</i> )
toAbsolutePath()	Devuelve la ruta absoluta del Path
fileName()	Devuelve el nombre del fichero o directorio final de la ruta

### Ejemplo 1:

```
import java.nio.file.Path

fun main() {
    // Path relativo al directorio del proyecto
    val rutaRelativa: Path = Path.of("documentos", "ejemplo.txt")
    // Path absoluto en Windows
    val rutaAbsolutaWin: Path = Path.of("C:", "Users", "Pol", "Documentos")
    // Path absoluto en Linux/macOS
    val rutaAbsolutaNix: Path = Path.of("/home/pol/documentos")

    println("Ruta relativa: " + rutaRelativa) // Muestra la ruta relativa
    println("Ruta absoluta: " + rutaRelativa.toAbsolutePath()) // ruta completa
    println("Ruta absoluta: " + rutaAbsolutaWin) // ruta absoluta Windows
    println("Ruta absoluta: " + rutaAbsolutaNix) // ruta absoluta Linux/macOS
}
```



Ejecuta el ejemplo anterior y comprueba que la salida es la siguiente:

```
Ruta relativa: documentos\ejemplo.txt
Ruta absoluta: F:\kot\1-ficheros\documentos\ejemplo.txt
Ruta absoluta: C:\Users\Pol\Documentos
Ruta absoluta: \home\pol\documentos
```

**Files:** Es una clase de utilidad con las acciones (borrar, copiar, mover, leer, etc) que podemos realizar sobre las rutas (Path). Algunos de sus principales métodos son:

Método	Descripción
exists(), isDirectory(), isRegularFile(), isReadable()	Verificar de existencia y accesibilidad
list(), walk()	Listar contenido de un directorio
readAttributes()	Obtener atributos (tamaño, fecha, etc.):
createDirectory()	Crear un directorio: Solo crea el directorio y espera que todo el "camino" hasta él ya exista
createDirectories	Crea un directorio y también los directorios padre si no existen. Es la forma más segura.
createFile()	Crear un fichero
delete()	Borrar un fichero o directorio (lanza una excepción si el borrado falla). Lanza la excepción <i>NoSuchFileException</i> si el fichero o directorio no existe. Es más seguro <i>deleteIfExists()</i>
move(origen, destino)	Mover o renombrar un fichero o directorio
copy(origen, destino)	Copiar un fichero o directorio. Si el destino ya existe se puede sobreescribir utilizando <i>copy(Path, Path, REPLACE_EXISTING)</i> . Si se copia un directorio no se copiará su contenido, el nuevo directorio estará vacío.

**Ejemplo 2:** El siguiente ejemplo es un organizador de ficheros. Imagina una carpeta de "multimedia" donde todo está desordenado. El programa organizará los ficheros en subcarpetas según su extensión (.pdf, .jpg, .mp3, etc).

```
import java.nio.file.Files
import java.nio.file.Path
import java.nio.file.StandardCopyOption
import kotlin.io.path.extension // Extensión de Kotlin para obtener la extensión

fun main() {
    // 1. Ruta de la carpeta a organizar
    val carpeta = Path.of("multimedia")
```

```

println("--- Iniciando la organización de la carpeta: " + carpeta + "---")
try {
    // 2. Recorrer la carpeta desordenada y utilizar .use para asegurar que los
    recursos del sistema se cierren correctamente
    Files.list(carpeta).use { streamDePaths ->
        streamDePaths.forEach { pathFichero ->
            // 3. Solo interesan los ficheros, ignorar subcarpetas
            if (Files.isRegularFile(pathFichero)) {
                // 4. Obteners la extensión del fichero (ej: "pdf", "jpg")
                val extension = pathFichero.extension.lowercase()
                if (extension.isBlank()) {
                    println("-> Ignorando: " + pathFichero.fileName)
                    return@forEach // Salta a la siguiente iteración del bucle
                }
                // 5. Crear la ruta del directorio de destino
                val carpetaDestino = carpeta.resolve(extension)
                // 6. Crear el directorio de destino si no existe
                if (!Files.exists(carpetaDestino)) {
                    println("-> Creando nueva carpeta " + extension)
                    Files.createDirectories(carpetaDestino)
                }
                // 7. Mover el fichero a su nueva carpeta
                val pathDestino = carpetaDestino.resolve(pathFichero.fileName)
                Files.move(pathFichero, pathDestino, StandardCopyOption.REPLACE_EXISTING)
                println("-> Moviendo " + pathFichero.fileName + " a " + extension)
            }
        }
    }
    println("\n--- ¡Organización completada con éxito! ---")
} catch (e: Exception) {
    println("\n--- Ocurrió un error durante la organización ---")
    e.printStackTrace()
}
}
}

```



Ejecuta el ejemplo anterior y comprueba que la salida es la siguiente:

```

--- Iniciando la organización de la carpeta: multimedia---
-> Creando nueva carpeta jpg
-> Moviendo 20191106_071048.jpg a jpg
-> Moviendo 20191101_071830.jpg a jpg
-> Creando nueva carpeta txt
-> Moviendo libros.txt a txt
-> Moviendo películas.txt a txt
-> Creando nueva carpeta pdf
-> Moviendo Lorem-ipsum-2.pdf a pdf

```

```
-> Moviendo Lorem-ipsum-1.pdf a pdf
-> Creando nueva carpeta mp3
-> Moviendo dark-cinematic-atmosphere.mp3 a mp3
-> Moviendo pad-harmonious-and-soothing-voice-like-background.mp3 a mp3

--- ¡Organización completada con éxito! ---
```

En el ejemplo anterior hemos recorrido un directorio para organizar los ficheros que contenía. Recorrer un directorio para "mirar" su contenido es útil en muchas situaciones y hay varias formas de hacerlo. A continuación veremos algunas:

**Files.list(path):** Es la utilizada en el ejemplo anterior. Lista únicamente el contenido de un directorio sin acceder a las subcarpetas. Será útil cuando solamente sea necesario acceder al contenido directo de una carpeta, por ejemplo para organizar ficheros en un directorio, mostrar el contenido de la carpeta actual o buscar un fichero específico solo en este nivel.

Ventajas:

- Rápido y eficiente al no ser recursivo.
- Ofrece un control preciso, operando solo en el primer nivel del directorio.
- Devuelve un Stream de Java que permite usar operadores funcionales (filter, map, etc.) de forma segura con .use.

Inconvenientes:

- No explora subdirectorios.
- Para recorrer un árbol completo, se necesita implementar lógica recursiva manualmente.

**Files.walk(path):** Recorre un directorio y todo su contenido recursivamente. Entra en cada subcarpeta, y en sus subcarpetas hasta el final. Será útil para operar sobre un directorio y todo lo que contiene, sin importar la profundidad, por ejemplo para buscar un fichero por nombre en cualquier subcarpeta, eliminar todos los ficheros temporales de un proyecto o contar todos los ficheros .kt de un repositorio.

Ventajas:

- Recorre árboles de directorios completos (recursivo) de forma muy sencilla.
- Extremadamente potente para búsquedas profundas o aplicar operaciones a todos los elementos anidados.

- También devuelve un Stream, permitiendo un filtrado y procesamiento muy expresivo.

Inconvenientes:

- Puede ser lento y consumir más memoria en directorios con miles de ficheros.
- Es una herramienta excesiva ('overkill') para tareas que solo requieren acceder al nivel actual.

**Files.newDirectoryStream(path)**: Es similar a Files.list(), listando solo el contenido inmediato. La diferencia es que no devuelve un Stream de Java 8 (que permite usar .filter, .forEach, etc.), sino un DirectoryStream, que es una versión más antigua que se usa con bucles for. Es menos común en código Kotlin moderno, pero es bueno reconocerlo para poder entender en proyectos antiguos (legacy). Para cualquier tarea nueva, Files.list() y Files.walk() son superiores en seguridad y expresividad.

Ventajas:

- Utiliza un bucle for-each tradicional, que puede resultar familiar.

Inconvenientes:

- ¡PELIGRO! Requiere cerrar el recurso manualmente (.close()). Si se olvida, provoca fugas de recursos (resource leaks).
- Es menos expresivo que los Streams. No se pueden encadenar operadores funcionales fácilmente.
- Considerado obsoleto en código Kotlin idiomático, que prefiere Files.list().use{...}.

**Ejemplo 3:** Queremos crear un informe de toda la estructura de la carpeta resultante del ejemplo anterior. Por tanto necesitamos entrar en las nuevas carpetas (pdf, jpg, txt) y ver qué ficheros hay dentro de cada una. Para ello se utiliza Files.walk() que calcula la profundidad, recorre la jerarquía de carpetas y muestra cada elemento indicando si es un directorio o un fichero.

```
import java.nio.file.Files
import java.nio.file.Path

fun main() {
    val carpetaPrincipal = Path.of("multimedia")

    println("--- Mostrando la estructura final con Files.walk() ---")
    try {
        Files.walk(carpetaPrincipal).use { stream ->
```

```
// Ordenar el stream para una visualización más predecible
stream.sorted().forEach { path ->
    // Calcular profundidad para la indentación
    // Restamos el número de componentes de la ruta base para que el
    directorio principal no tenga indentación
    val profundidad = path.nameCount - carpetaPrincipal.nameCount
    val indentacion = "\t".repeat(profundidad)

    // Determinamos si es directorio o fichero para el prefijo
    val prefijo = if (Files.isDirectory(path)) "[DIR]" else "[FILE]"

    // No imprimimos la propia carpeta raíz, solo su contenido
    if (profundidad > 0) {
        println("$indentacion$prefijo ${path.fileName}")
    }
}
} catch (e: Exception) {
    println("\n--- Ocurrió un error durante el recorrido ---")
    e.printStackTrace()
}
}
```



Ejecuta el ejemplo anterior y comprueba que la salida es la siguiente:

```
--- Mostrando la estructura final con Files.walk() ---
[DIR] jpg
    [FILE] 20191101_071830.jpg
    [FILE] 20191106_071048.jpg
[DIR] mp3
    [FILE] dark-cinematic-atmosphere.mp3
    [FILE] pad-harmonious-and-soothing-voice-like-background.mp3
[DIR] mp4
    [FILE] 283533_small.mp4
    [FILE] 293968_small.mp4
[DIR] pdf
    [FILE] lorem-ipsum-1.pdf
    [FILE] lorem-ipsum-2.pdf
[DIR] txt
    [FILE] libros.txt
    [FILE] peliculas.txt
```

## 🎯 Práctica 2: Directorios y comprobaciones

Elimina el código de ejemplo del proyecto y escribe las líneas necesarias para que tu proyecto haga lo siguiente:

1. **Defina dos rutas:** una para una carpeta llamada `datos_ini` y otra para una carpeta llamada `datos_fin` (ambas dentro de la carpeta de tu proyecto).
2. **Compruebe directorios:** Si las carpetas no existen las deberá crear utilizando `Files.createDirectories`.
3. **Compruebe ficheros:** Despues de la comprobación de la existencia de un fichero de datos dentro de la carpeta `datos_ini` (por ejemplo `mis_datos.json`) imprimirá un mensaje por consola indicando si lo ha encontrado o no.

## 3. Ficheros de texto

Los ficheros de texto son legibles directamente por humanos y son una buena opción para guardar información después de cerrar el programa. A continuación se muestran algunas clases y métodos para leer y escribir información en ellos:

Método	Descripción
<code>Files.readAllLines(path)</code> devuelve <code>List&lt;String&gt;</code>	Leer ficheros
<code>Files.exists(path)</code>	Verificar existencia
<code>split(), trim(), toIntOrNull()</code>	Procesar texto
<code>Files.write(path, lines)</code>	Escribe una lista de líneas ( <code>List&lt;String&gt;</code> ) a un fichero.
<code>StandardOpenOption.READ</code>	Abrir un fichero en modo lectura.
<code>StandardOpenOption.WRITE</code>	Abrir un fichero en modo escritura.
<code>StandardOpenOption.APPEND</code>	Agrega contenido al final del fichero sin borrar lo anterior.
<code>StandardOpenOption.CREATE</code>	Si no existe, lo crea.
<code>StandardOpenOption.TRUNCATE_EXISTING</code>	Si existe, borra lo anterior.
<code>Files.newBufferedReader(Path)</code> <code>Files.newBufferedWriter(Path)</code>	Más eficiente para ficheros grandes

Files.readString(Path) (Java 11+) Files.writeString(Path, String)	Lectura/escritura completa como bloque
--	--

Dentro de los ficheros de texto existen ficheros de texto plano (sin ningún tipo de estructura) y ficheros de texto en los que la información está estructurada.

#### Ejemplo 4: Escritura y lectura de ficheros de texto plano .txt

```
import java.nio.file.Files
import java.nio.file.Paths
import java.nio.charset.StandardCharsets

fun main() {
    //Escritura en fichero de texto
    //writeString
    val texto = "Hola, mundo desde Kotlin"
    Files.writeString(Paths.get("documentos/saludo.txt"), texto)

    //write
    val ruta = Paths.get("documentos/texto.txt")
    val lineasParaGuardar = listOf(
        "Primera línea",
        "Segunda línea",
        "¡Hola desde Kotlin!"
    )
    Files.write(ruta, lineasParaGuardar, StandardCharsets.UTF_8)
    println("Fichero de texto escrito.")

    //newBuffered
    Files.newBufferedWriter(Paths.get("documentos/log.txt")).use { writer
        ->
        writer.write("Log iniciado...\n")
        writer.write("Proceso completado.\n")

        //Lectura del fichero de texto

        //readAllLines
        val lineasLeidas = Files.readAllLines(ruta)
        println("Contenido leído con readAllLines:")
        for (lineas in lineasLeidas) {
            println(lineas)
        }
    }
}
```

```
//readString
val contenido = Files.readString(ruta)
println("Contenido leído con readString:")
println(contenido)

//newBufferedReader
Files.newBufferedReader(ruta).use { reader ->
    println("Contenido leído con newBufferedReader:")
    reader.lineSequence().foreach { println(it) }
}
}
```

Ejecuta el ejemplo anterior y comprueba que la salida es la siguiente:

## *Fichero de texto escrito.*

Contenido leído con readAllLines:

Primera Línea

Segunda Línea

Segunda etapa  
*Hola desde Kotlin!*

Contenido leído con readString:

#### *Primera Línea*

*Primera Línea*  
*Segunda Línea*

## Segunda etapa

Contenido Leido con newBufferedReader:

### *Primera Línea*

Primera Etapa

## Segunda etapa

## 4. Ficheros de intercambio de información

Los ficheros de texto en los que la información está estructurada y organizada de una manera predecible permiten que distintos sistemas la lean y entiendan. Estos tipos de ficheros se utilizan en el desarrollo de software para intercambiar información entre aplicaciones y algunos de los formatos más importantes son CSV, JSON y XML.

Para poder llevar a cabo este intercambio de información, hay que extraer la información del fichero origen. Este proceso no se realiza línea por línea, sino que el contenido del fichero se lee (parsea) utilizando la técnica de **serialización/deserialización**:

- **Serialización:** Proceso de **convertir un objeto en memoria** (por ejemplo, una data class) en una representación textual o binaria (como un String en formato JSON o XML) que se puede guardar en un fichero o enviar por red.
- **Deserialización:** Es el proceso inverso de **leer un fichero** (JSON, XML, etc.) y **reconstruir el objeto original** en memoria para poder trabajar con él.

A continuación se muestra una tabla con clases y herramientas que se utilizan para serializar / deserializar:

Método	Descripción
java.io.Serializable	Marca que un objeto es serializable
ObjectOutputStream	Serializa y escribe un objeto
ObjectInputStream	Lee un objeto serializado
@transient	Excluye atributos de la serialización
ReadObject	Lee y reconstruye un objeto binario
WriteObject	Guarda un objeto como binario
@Serializable	Permite convertir el data class a JSON y viceversa.

### Ejemplo 5: Serializar y deserializar un objeto (usando @Transient)

```
import java.io.*

// Clase Persona (serializable completamente)
class Persona(val nombre: String, val edad: Int) : Serializable
// Clase Usuario con un atributo que NO se serializa
class Usuario(
    val nombre: String,
    @Transient val clave: String // Este campo no se guardará
) : Serializable
```

```

fun main() {
    val rutaPersona = "documentos/persona.obj"
    val rutaUsuario = "documentos/usuario.obj"

    // Asegurar que el directorio exista
    val directorio = File("documentos")
    if (!directorio.exists()) {
        directorio.mkdirs()
    }

    // --- Serializar Persona ---
    val persona = Persona("Pol", 30)
    try {
        ObjectOutputStream(FileOutputStream(rutaPersona)).use { oos ->
            oos.writeObject(persona)
        }
        println("Persona serializada.")
    } catch (e: IOException) {
        println("Error al serializar Persona: ${e.message}")
    }

    // --- Deserializar Persona ---
    try {
        val personaLeida = ObjectInputStream(FileInputStream(rutaPersona)).use { ois ->
            ois.readObject() as Persona
        }
        println("Persona deserializada:")
        println("Nombre: ${personaLeida.nombre}, Edad: ${personaLeida.edad}")
    } catch (e: Exception) {
        println("Error al deserializar Persona: ${e.message}")
    }

    // --- Serializar Usuario ---
    val usuario = Usuario("Eli", "1234")
    try {
        ObjectOutputStream(FileOutputStream(rutaUsuario)).use { oos ->
            oos.writeObject(usuario)
        }
        println("Usuario serializado.")
    } catch (e: IOException) {
        println("Error al serializar Usuario: ${e.message}")
    }

    // --- Deserializar Usuario ---
    try {
        val usuarioLeido = ObjectInputStream(FileInputStream(rutaUsuario)).use {
            ois ->
            ois.readObject() as Usuario
        }
    }
}

```

```

    }
    println("Usuario deserializado:")
    println("Nombre: ${usuarioLeido.nombre}, Clave: ${usuarioLeido.clave}")
} catch (e: Exception) {
    println("Error al deserializar Usuario: ${e.message}")
}
}
}

```



Ejecuta el ejemplo anterior y comprueba que la salida es la siguiente:

Persona serializada.  
 Persona deserializada:  
 Nombre: Pol, Edad: 30  
 Usuario serializado.  
 Usuario deserializado:  
 Nombre: Eli, Clave: null

A continuación se describen los 3 tipos de ficheros más comunes para intercambio de información. Se muestran ejemplos de lectura y escritura usando serialización y deserialización utilizando un proyecto con Gradle:

## 4.1. CSV (Comma-Separated Values)

Son ficheros de texto plano con valores separados por un delimitador (coma, punto y coma, etc.). Son útiles para exportar/importar datos desde Excel, Google Sheets, o bases de datos. Se manejan con herramientas como OpenCSV (más antigua) o **Kotlin-CSV** (la que utilizaremos). Algunos de sus métodos son:

Método	Ejemplo
readAll(File)	val filas = csvReader().readAll(File("alumnos.csv"))
readAllWithHeader(File)	val datos = csvReader().readAllWithHeader(File("alumnos.csv"))
open { readAllAsSequence() }	csvReader().open("alumnos.csv") { readAllAsSequence().forEach { println(it) } }
writeAll(data, File)	csvWriter().writeAll(listOf(listOf("Pol", "9")), File("salida.csv"))
writeRow(row, File)	csvWriter().writeRow(listOf("Ade", "8"), File("salida.csv"))

writeAllWithHeader(data, File)	csvWriter().writeAllWithHeader(listOf("nombre" to "Eli", "nota" to "10")), File("salida.csv"))
delimiter, quoteChar, etc.	csvReader { delimiter = ';' }

**Ejemplo 6:** Lectura y escritura de ficheros CSV. Partimos de un fichero llamado *mis\_plantas.csv* con la información siguiente:

```
1;Aloe Vera;Aloe barbadensis miller;7;0.6
2;Lavanda;Lavandula angustifolia;3;1.0
3;Helecho de Boston;Nephrolepis exaltata;5;0.9
4;Bambú de la suerte;Dracaena sanderiana;4;1.5
5;Girasol;Helianthus annuus;2;3.0
```

Donde los campos corresponden a:

- id\_planta (int)
- nombre\_comun (string)
- nombre\_cientifico (string)
- frecuencia\_riego (int)
- altura\_máxima (double)

Utilizaremos la librería **Kotlin-CSV**. Por tanto habrá que indicarlo en el fichero *build.gradle.kts* añadiendo las siguientes líneas:

- En plugins: `kotlin("plugin.serialization") version "1.9.0"`
- En dependencies:

```
implementation("com.github.doyaaka:kotlin-csv-jvm:1.9.1")
```

```
import java.nio.file.Files
import java.nio.file.Path
import java.io.File
// Librería específica de Kotlin para Leer y escribir ficheros CSV.
import com.github.doyaaka.kotlinCsv.dsl.csvReader
import com.github.doyaaka.kotlinCsv.dsl.csvWriter

//Usamos una 'data class' para representar la estructura de una planta.
data class Planta(val id_planta: Int, val nombre_comun: String, val
nombre_cientifico: String, val riego: Int, val altura: Double)

fun main() {
    val entradaCSV = Path.of("datos_ini/mis_plantas.csv")
    val salidaCSV = Path.of("datos_ini/mis_plantas2.csv")
    val datos: List<Planta>
    datos = LeerDatosInicialesCSV(entradaCSV)
    for (dato in datos) {
```

```

        println(" - ID: ${dato.id_planta}, Nombre común: ${dato.nombre_comun},
Nombre científico: ${dato.nombre_cientifico}, Frecuencia de riego: ${dato.riego}
días, Altura: ${dato.altura} metros")
    }
    escribirDatosCSV(salidaCSV, datos)
}

fun leerDatosInicialesCSV(ruta: Path): List<Planta>
{
    var plantas: List<Planta> =emptyList()
    // Comprobar si el fichero es legible antes de intentar procesarlo.
    if (!Files.isReadable(ruta)) {
        println("Error: No se puede leer el fichero en la ruta: $ruta")
    } else{
        // Configuramos el lector de CSV con el delimitador
        val reader = csvReader {
            delimiter = ';'
        }
        /* Leemos TODO el fichero CSV.
        El resultado es una Lista de Listas de Strings (`List<List<String>>`),
        donde cada Lista interna representa una fila del fichero.*/
        val filas: List<List<String>> = reader.readAll(ruta.toFile())

        /* Convertir la lista de texto plano en una lista de objetos 'Planta'.
        `mapNotNull` funciona como un `map` y descartando todos los `null` de la
        lista final. Si una fila del CSV es inválida, devolvemos `null`
        y `mapNotNull` se encarga de ignorarla. */
        plantas = filas.mapNotNull { columnas ->
            // Validar si la fila tiene al menos 4 columnas.
            if (columnas.size >= 5) {
                try {
                    val id_planta = columnas[0].toInt()
                    val nombre_comun = columnas[1]
                    val nombre_cientifico = columnas[2]
                    val riego = columnas[3].toInt()
                    val altura = columnas[4].toDouble()
                    Planta(id_planta, nombre_comun, nombre_cientifico, riego, altura)
                } catch (e: Exception) {
                    /* Si ocurre un error en la conversión (ej: NumberFormatException),
                    capturamos la excepción, imprimimos un aviso (opcional)
                    y devolvemos `null` para que `mapNotNull` descarte esta fila. */
                    println("Fila inválida ignorada: $columnas -> Error: ${e.message}")
                    null
                }
            } else {
                // La fila no tiene suficientes columnas, es inválida. Devolver null.
                println("Fila con formato incorrecto ignorada: $columnas")
            }
        }
    }
}

```

```

        null
    }
}
}
return plantas
}

fun escribirDatosCSV(ruta: Path, plantas: List<Planta>){
    try {
        val fichero: File = ruta.toFile()

        csvWriter {
            delimiter = ';'
        }.writeAll(
            plantas.map { planta ->
                listOf(planta.id_planta.toString(),
                    planta.nombre_comun,
                    planta.nombre_cientifico,
                    planta.riego.toString(),
                    planta.altura.toString())
            },
            fichero
        )
        println("\nInformación guardada en: $fichero")
    } catch (e: Exception) {
        println("Error: ${e.message}")
    }
}
}
```

🔍 Ejecuta el ejemplo anterior, comprueba que la salida es la siguiente, que se ha creado el fichero *mis\_plantas2.csv* y que su contenido es correcto:

- ID: 1, Nombre común: Aloe Vera, Nombre científico: *Aloe barbadensis miller*, Frecuencia de riego: 7 días, Altura: 0.6 metros
- ID: 2, Nombre común: Lavanda, Nombre científico: *Lavandula angustifolia*, Frecuencia de riego: 3 días, Altura: 1.0 metros
- ID: 3, Nombre común: Helecho de Boston, Nombre científico: *Nephrolepis exaltata*, Frecuencia de riego: 5 días, Altura: 0.9 metros
- ID: 4, Nombre común: Bambú de la suerte, Nombre científico: *Dracaena sanderiana*, Frecuencia de riego: 4 días, Altura: 1.5 metros
- ID: 5, Nombre común: Girasol, Nombre científico: *Helianthus annuus*, Frecuencia de riego: 2 días, Altura: 3.0 metros

Información guardada en: *datos\_ini\mis\_plantas2.csv*

### 🎯 Práctica 3: Creación de tu CSV

Realiza lo siguiente:

1. **Diseña tu data class:** Define la data class de Kotlin que represente un único elemento de tu colección de datos. Debe tener un ID (Int), un nombre (String) y, al menos, otros tres campos (uno de tipo Double).
2. **Crea tu fichero de datos** con al menos 5 registros de tu colección dentro de la carpeta *datos\_ini*.
3. **Replica el ejemplo anterior** para que funcione con tu archivo de datos.

#### Aspectos Técnicos Obligatorios:

- **Añade dependencias necesarias:** Añade las librerías necesarias para leer tu fichero y serializar / deserializar datos en *build.gradle.kts*
- Se debe incluir un manejo **básico de errores** (ej: comprobar si el fichero existe antes de leerlo, try-catch para conversiones numéricas, etc.).

## 4.2. XML (eXtensible Markup Language)

Los ficheros XML son muy estructurados y extensibles. Se basan en etiquetas anidadas similar a HTML. Permiten la validación de datos (mediante esquemas XSD) y es ideal para integración con sistemas empresariales (legacy). Se manejan con librerías como JAXB, DOM, JDOM2 o **Jackson XML (XmlMapper)** que es la que utilizaremos. Algunos de sus métodos son:

Método	Descripción
readValue(File, Class<T>)	Lee un fichero XML y lo convierte en un objeto Kotlin/Java
readValue(String, Class<T>)	Lee un String XML y lo convierte en un objeto
writeValue(File, Object)	Escribe un objeto como XML en un fichero
writeValueAsString(Object)	Convierte un objeto en una cadena XML
writeValueAsBytes(Object)	Convierte un objeto en un array de bytes XML
registerModule(Module)	Registra un módulo como KotlinModule o JavaTimeModule

enable(SerializationFeature)	Activa una opción de serialización (por ejemplo, indentado)
disable(DeserializationFeature)	Desactiva una opción de deserialización
configure(MapperFeature, boolean)	Configura opciones generales del mapeo
setDefaultPrettyPrinter(...)	Establece un formateador personalizado

**Ejemplo 7:** Lectura y escritura de ficheros XML. Partimos de un fichero llamado *mis\_plantas.xml* con la información siguiente:

```
<plantas>
  <planta>
    <id_planta>1</id_planta>
    <nombre_comun>Aloe Vera</nombre_comun>
    <nombre_cientifico>Aloe barbadensis miller</nombre_cientifico>
    <frecuencia_riego>7</frecuencia_riego>
    <altura_maxima>0.6</altura_maxima>
  </planta>
  <planta>
    <id_planta>2</id_planta>
    <nombre_comun>Lavanda</nombre_comun>
    <nombre_cientifico>Lavandula angustifolia</nombre_cientifico>
    <frecuencia_riego>3</frecuencia_riego>
    <altura_maxima>1.0</altura_maxima>
  </planta>
  <planta>
    <id_planta>3</id_planta>
    <nombre_comun>Helecho de Boston</nombre_comun>
    <nombre_cientifico>Nephrolepis exaltata</nombre_cientifico>
    <frecuencia_riego>5</frecuencia_riego>
    <altura_maxima>0.9</altura_maxima>
  </planta>
  <planta>
    <id_planta>4</id_planta>
    <nombre_comun>Bambú de la suerte</nombre_comun>
    <nombre_cientifico>Dracaena sanderiana</nombre_cientifico>
    <frecuencia_riego>4</frecuencia_riego>
    <altura_maxima>1.5</altura_maxima>
  </planta>
  <planta>
    <id_planta>5</id_planta>
    <nombre_comun>Girasol</nombre_comun>
    <nombre_cientifico>Helianthus annuus</nombre_cientifico>
```

```

<frecuencia_riego>2</frecuencia_riego>
<altura_maxima>3.0</altura_maxima>
</planta>
</plantas>
```

Utilizaremos la librería **Jackson XML**. Por tanto habrá que indicarlo en el fichero *build.gradle.kts* añadiendo las siguientes líneas:

```

implementation("com.fasterxml.jackson.dataformat:jackson-dataformat-xml:2.17.0")
implementation("com.fasterxml.jackson.module:jackson-module-kotlin:2.17.0")

import java.nio.file.Path
import java.io.File

// Anotaciones y clases de la librería Jackson para el mapeo a XML.
import com.fasterxml.jackson.dataformat.xml.XmlMapper
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlRootElement
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlElementWrapper
import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlProperty
import com.fasterxml.jackson.module.kotlin.readValue
import com.fasterxml.jackson.module.kotlin.registerKotlinModule

/*Representa la estructura de una única planta. La propiedad 'id_planta' será la
etiqueta <id_planta>...</id_planta> (así todas) */
data class Planta(
    @JacksonXmlProperty(localName = "id_planta")
    val id_planta: Int,
    @JacksonXmlProperty(localName = "nombre_comun")
    val nombre_comun: String,
    @JacksonXmlProperty(localName = "nombre_cientifico")
    val nombre_cientifico: String,
    @JacksonXmlProperty(localName = "frecuencia_riego")
    val frecuencia_riego: Int,
    @JacksonXmlProperty(localName = "altura_maxima")
    val altura_maxima: Double
)

//nombre del elemento raíz
@JacksonXmlRootElement(localName = "plantas")

// Data class que representa el elemento raíz del XML.
data class Plantas(
    @JacksonXmlElementWrapper(useWrapping = false) // No necesitamos la etiqueta
<plantas> aquí
    @JacksonXmlProperty(localName = "planta")
    val listaPlantas: List<Planta> = emptyList()
)
```

```

fun main() {
    val entradaXML = Path.of("datos_ini/mis_plantas.xml")
    val salidaXML = Path.of("datos_ini/mis_plantas2.xml")

    val datos: List<Planta>
    datos = leerDatosInicialesXML(entradaXML)
    for (dato in datos) {
        println(" - ID: ${dato.id_planta}, Nombre común: ${dato.nombre_comun},
Nombre científico: ${dato.nombre_cientifico}, Frecuencia de riego:
${dato.frecuencia_riego} días, Altura: ${dato.altura_maxima} metros")
    }
    escribirDatosXML(salidaXML, datos)
}

fun leerDatosInicialesXML(ruta: Path): List<Planta> {
//var plantas: List<Planta> =emptyList()
    val fichero: File = ruta.toFile()

    // Deserializar el XML a objetos Kotlin
    val xmlMapper = XmlMapper().registerKotlinModule()
    // 'readValue' convierte el contenido XML en una instancia de la clase 'Plantas'
    val plantasWrapper: Plantas = xmlMapper.readValue(fichero)
    return plantasWrapper.listaPlantas
}

fun escribirDatosXML(ruta: Path, plantas: List<Planta>) {
    try {
        val fichero: File = ruta.toFile()

        // Creamos instancia de la clase 'Plantas' (raíz del XML).
        val contenedorXml = Plantas(plantas)

        // Configuramos el 'XmlMapper' (motor de Jackson) para la conversión a XML.
        val xmlMapper = XmlMapper().registerKotlinModule()

        // Convertimos 'contenedorXml' en un String con formato XML.
        // .writerWithDefaultPrettyPrinter() formatea con indentación y saltos de
        Línea
        val xmlString =
        xmlMapper.writerWithDefaultPrettyPrinter().writeValueAsString(contenedorXml)

        // escribir un String en un fichero con 'writeText'
        fichero.writeText(xmlString)

        println("\nInformación guardada en: $fichero")
    } catch (e: Exception) {
        println("Error: ${e.message}")
    }
}

```

}

}

 Ejecuta el ejemplo anterior, comprueba que la salida es la siguiente, que se ha creado el fichero *mis\_plantas2.xml* y que su contenido es correcto:

- *ID: 1, Nombre común: Aloe Vera, Nombre científico: Aloe barbadensis miller, Frecuencia de riego: 7 días, Altura: 0.6 metros*
- *ID: 2, Nombre común: Lavanda, Nombre científico: Lavandula angustifolia, Frecuencia de riego: 3 días, Altura: 1.0 metros*
- *ID: 3, Nombre común: Helecho de Boston, Nombre científico: Nephrolepis exaltata, Frecuencia de riego: 5 días, Altura: 0.9 metros*
- *ID: 4, Nombre común: Bambú de La suerte, Nombre científico: Dracaena sanderiana, Frecuencia de riego: 4 días, Altura: 1.5 metros*
- *ID: 5, Nombre común: Girasol, Nombre científico: Helianthus annuus, Frecuencia de riego: 2 días, Altura: 3.0 metros*

Información guardada en: *datos\_ini\mis\_plantas2.xml*

## Práctica 4: Creación de tu XML

Realiza lo siguiente:

1. **Diseña tu data class:** Puedes utilizar la misma que en la práctica anterior.
2. **Crea tu fichero de datos** con al menos 5 registros de tu colección dentro de la carpeta *datos\_ini*.
3. **Replica el ejemplo anterior** para que funcione con tu archivo de datos.

### Aspectos Técnicos Obligatorios:

- **Añade dependencias necesarias:** Añade las librerías necesarias para leer tu fichero y serializar / deserializar datos en *build.gradle.kts*
- Se debe incluir un manejo **básico de errores** (ej: comprobar si el fichero existe antes de leerlo, try-catch para conversiones numéricas, etc.).

## 4.3. JSON (JavaScript Object Notation)

Son ficheros ligeros, fáciles de leer y con una estructura de pares clave-valor y listas. Ideales para APIs REST, ficheros de configuración y bases de datos NoSQL

(como MongoDB). Se maneja con librerías como Jackson & Gson (Java) o **kotlinx.serialization** (la que utilizaremos en Kotlin). Algunos de sus métodos son:

Método / Ejemplo	Descripción
<code>Json.encodeToString(objeto)</code> <code>Json.encodeToString(persona)</code>	Convierte un objeto Kotlin a una cadena JSON
<code>Json.encodeToString(serializer, obj)</code> <code>Json.encodeToString(Persona.serializer(), persona)</code>	Igual que el anterior pero especificando el serializador
<code>Json.decodeFromString(json)</code> <code>Json.decodeFromString&lt;Persona&gt;(json)</code>	Convierte una cadena JSON a un objeto Kotlin
<code>Json.decodeFromString(serializer, s)</code> <code>Json.decodeFromString(Persona.serializer(), json)</code>	Igual que el anterior pero con el serializador explícito
<code>Json.encodeToJsonElement(objeto)</code> <code>val elem = Json.encodeToJsonElement(persona)</code>	Convierte un objeto a un árbol JsonElement
<code>Json.decodeFromJsonElement(elem)</code> <code>val persona = Json.decodeFromJsonElement&lt;Persona&gt;(elem)</code>	Convierte un JsonElement a objeto Kotlin
<code>Json.parseToJsonElement(string)</code> <code>val elem = Json</code>	Parsea una cadena JSON a un árbol JsonElement sin mapear

**Ejemplo 8:** Lectura y escritura de ficheros JSON. Partimos de un fichero llamado *mis\_plantas.json* con la información siguiente:

```
[  {
    "id_planta": 1,
    "nombre_comun": "Aloe Vera",
    "nombre_cientifico": "Aloe barbadensis miller",
    "frecuencia_riego": 7,
    "altura_maxima": 0.6
},
{
    "id_planta": 2,
    "nombre_comun": "Lavanda",
    "nombre_cientifico": "Lavandula angustifolia",
    "frecuencia_riego": 3,
    "altura_maxima": 1.0
},
```

```
{
    "id_planta": 3,
    "nombre_comun": "Helecho de Boston",
    "nombre_cientifico": "Nephrolepis exaltata",
    "frecuencia_riego": 5,
    "altura_maxima": 0.9
},
{
    "id_planta": 4,
    "nombre_comun": "Bambú de la suerte",
    "nombre_cientifico": "Dracaena sanderiana",
    "frecuencia_riego": 4,
    "altura_maxima": 1.5
},
{
    "id_planta": 5,
    "nombre_comun": "Girasol",
    "nombre_cientifico": "Helianthus annuus",
    "frecuencia_riego": 2,
    "altura_maxima": 3.0
} ]
```

Utilizaremos la librería **kotlinx.serialization**. Por tanto habrá que indicarlo en el fichero *build.gradle.kts* añadiendo las siguientes líneas:

- En plugins

```
kotlin("plugin.serialization") version "1.9.0"
```

- En dependencies:

```
implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.6.0")
```

Llamaremos a **Json.encodeToString()** para serializar una instancia de esta clase y a **Json.decodeFromString()** para deserializarla.

```
import java.nio.file.Files
import java.nio.file.Path
import java.io.File

// Clases de la librería oficial de Kotlin para la serialización/deserialización.
import kotlinx.serialization.*
import kotlinx.serialization.json.*

// Usamos una 'data class' para representar la estructura de una planta e indicamos
// que es serializable
@Serializable
data class Planta(val id_planta: Int, val nombre_comun: String, val
nombre_cientifico: String, val frecuencia_riego: Int, val altura_maxima: Double)
```

```

fun main() {
    val entradaJSON = Path.of("datos_ini/mis_plantas.json")
    val salidaJSON = Path.of("datos_ini/mis_plantas2.json")
    val datos: List<Planta>
    datos = LeerDatosInicialesJSON(entradaJSON)
    for (dato in datos) {
        println(" - ID: ${dato.id_planta}, Nombre común: ${dato.nombre_comun},
Nombre científico: ${dato.nombre_cientifico}, Frecuencia de riego:
${dato.frecuencia_riego} días, Altura: ${dato.altura_maxima} metros")
    }
    escribirDatosJSON(salidaJSON, datos)
}

fun LeerDatosInicialesJSON(ruta: Path): List<Planta> {
    var plantas: List<Planta> = emptyList()
    val jsonString = Files.readString(ruta)

    /* A `Json.decodeFromString` le pasamos el String con el JSON.
       Con `<List<Planta>>`, le indicamos que debe interpretarlo como
       una lista de objetos de tipo planta».
       La librería usará la anotación @Serializable de la clase Planta para saber
       cómo mapear los campos del JSON ("id_planta", "nombre_comun", etc.)
       a las propiedades del objeto. */
    plantas = Json.decodeFromString<List<Planta>>(jsonString)
    return plantas
}

fun escribirDatosJSON(ruta: Path, plantas: List<Planta>) {
    try {
        /* La librería `kotlinx.serialization`
           toma la lista de objetos `Planta` (`List<Planta>`) y la convierte en una
           única cadena de texto con formato JSON.
           `prettyPrint` formatea el JSON para que sea legible. */
        val json = Json { prettyPrint = true }.encodeToString(plantas)

        // Con `Files.writeString` escribimos el String JSON en el fichero de salida
        Files.writeString(ruta, json)

        println("\nInformación guardada en: $ruta")
    } catch (e: Exception) {
        println("Error: ${e.message}")
    }
}

```



Ejecuta el ejemplo anterior, comprueba que la salida es la siguiente, que se

ha creado el fichero *mis\_plantas2.json* y que su contenido es correcto:

- ID: 1, Nombre común: Aloe Vera, Nombre científico: *Aloe barbadensis miller*, Frecuencia de riego: 7 días, Altura: 0.6 metros
- ID: 2, Nombre común: Lavanda, Nombre científico: *Lavandula angustifolia*, Frecuencia de riego: 3 días, Altura: 1.0 metros
- ID: 3, Nombre común: Helecho de Boston, Nombre científico: *Nephrolepis exaltata*, Frecuencia de riego: 5 días, Altura: 0.9 metros
- ID: 4, Nombre común: Bambú de La suerte, Nombre científico: *Dracaena sanderiana*, Frecuencia de riego: 4 días, Altura: 1.5 metros
- ID: 5, Nombre común: Girasol, Nombre científico: *Helianthus annuus*, Frecuencia de riego: 2 días, Altura: 3.0 metros

Información guardada en: *datos\_ini\mis\_plantas2.json*

### 🎯 Práctica 5: Creación de tu JSON

Realiza lo siguiente:

1. **Diseña tu data class:** Puedes utilizar la misma que en la práctica anterior.
2. **Crea tu fichero de datos** con al menos 5 registros de tu colección dentro de la carpeta *datos\_ini*.
3. **Replica el ejemplo anterior** para que funcione con tu archivo de datos.

#### Aspectos Técnicos Obligatorios:

- **Añade dependencias necesarias:** Añade las librerías necesarias para leer tu fichero y serializar / deserializar datos en *build.gradle.kts*
- Se debe incluir un manejo **básico de errores** (ej: comprobar si el fichero existe antes de leerlo, try-catch para conversiones numéricas, etc.).

## 4.4. Conversiones entre ficheros

Una vez vistas las características de los ficheros de intercambio de información más comunes podemos llegar a la conclusión que en programación y gestión de datos, no todos los formatos sirven igual para todos los casos. **Convertir entre CSV, JSON y XML** permite aprovechar las ventajas de cada uno.

El patrón para convertir datos de un formato a otro es casi siempre el mismo. En lugar de intentar una conversión directa, utilizamos nuestras clases de Kotlin (data class) como un paso intermedio universal:

**Formato Origen → Objetos Kotlin en Memoria → Formato Destino**

## 🎯 Práctica 6: Elige tu fichero de datos

Realiza lo siguiente:

1. **Elige tu fichero de datos**: (.csv, .json o .xml) a partir de los creados anteriormente.
2. **Convierte tu fichero a otro formato** (.csv, .json o .xml)
3. Crea una función que lea el fichero resultante de la conversión y devuelva una lista de objetos, por ejemplo: **leerDatosIniciales(): List<DataClass>**.
4. **Verifica que funciona**: Imprime por consola la información leída.

### Aspectos Técnicos Obligatorios:

- Se debe incluir un manejo **básico de errores** (ej: comprobar si el fichero existe antes de leerlo, try-catch para conversiones numéricas, etc.).

## 📁 Entrega parcial

Entrega el código fuente del proyecto comprimido en un fichero .zip para que la profesora te dé sugerencias de mejora (el programa entregado deberá ejecutarse, si da error de ejecución, no se podrá revisar).

Hasta ahora tu programa debe cumplir los siguientes aspectos:

- Estar correctamente configurado con Gradle.
- Tener el código bien estructurado con una única función main y el resto de funciones legibles y con nombres claros.
- Utilizar las clases java.nio.file.Path y java.nio.file.Files para gestionar rutas.
- Gestionar adecuadamente las excepciones de E/S (IOException, FileNotFoundException).
- Definir una data class para representar un registro de información.
- Leer y escribir correctamente los ficheros de CSV, XML y JSON..
- Realizar conversiones entre los diferentes formatos de archivos anteriores.

## 5. Ficheros binarios

Los ficheros binarios no son legibles directamente por humanos (.exe, .jpg, .mp3, .dat). En ellos los datos pueden estar no estructurados o estructurados.

## 5.1. Ficheros binarios no estructurados

En los ficheros binarios no estructurados los datos se escriben “tal cual” en bytes, sin un formato estructurado definido por un estándar. El programa que los lee necesita saber cómo interpretar esos bytes.

Método	Descripción
Files.readAllBytes(Path) Files.write(Path, ByteArray)	Lee y escribe bytes puros
Files.newInputStream(Path) Files.newOutputStream(Path)	Flujo de bytes directo

**Ejemplo 9:** Escritura bit a bit de los datos 1 2 3 4 5 en un fichero binario no estructurado llamado *datos.bin*

```
import java.io.IOException
import java.nio.file.Files
import java.nio.file.Path

fun main() {
    val ruta = Path.of("multimedia/bin/datos.bin")

    try {
        // Asegura que el directorio existe
        val directorio = ruta.parent
        if (directorio != null && !Files.exists(directorio)) {
            Files.createDirectories(directorio)
            println("Directorio creado: ${directorio.toAbsolutePath()}")
        }

        // Verifica si se puede escribir
        if (!Files.isWritable(directorio)) {
            println("No se tienen permisos de escritura en el directorio: $directorio")
        } else {
            // Datos a escribir
            val datos = byteArrayOf(1, 2, 3, 4, 5)
            Files.write(ruta, datos)
            println("Fichero binario creado: ${ruta.toAbsolutePath()}")
        }

        // Verifica si se puede leer
        if (!Files.isReadable(ruta)) {
            println("No se tienen permisos de lectura para el fichero: $ruta")
        } else {
            // Lectura del fichero binario
            val bytes = Files.readAllBytes(ruta)
        }
    } catch (e: IOException) {
        e.printStackTrace()
    }
}
```

```

        println("Contenido leído (byte a byte):")
        for (b in bytes) {
            print("$b ")
        }
    }
} catch (e: IOException) {
    println("Ocurrió un error de entrada/salida: ${e.message}")
} catch (e: SecurityException) {
    println("No se tienen permisos suficientes: ${e.message}")
} catch (e: Exception) {
    println("Error inesperado: ${e.message}")
}
}
}

```



Ejecuta el ejemplo anterior y comprueba que la salida es la siguiente:

```

Directorio creado: F:\kot\1-ficheros\ejemplos1\multimedia\bin
Fichero binario creado: F:\kot\1-ficheros\ejemplos1\multimedia\bin\datos.bin
Contenido Leído (byte a byte):
1 2 3 4 5

```

## 5.2. Ficheros binarios estructurados

En los ficheros binarios estructurados los datos se guardan de forma estructurada siguiendo una organización predefinida, con campos y tipos de datos, a veces con un formato estándar (ej. PNG, ZIP, MP3, etc.) que pueden incluir cabeceras (con información como versión, tamaño, etc.) y registros (con campos fijos o delimitadores). El orden de bytes y los tamaños están definidos, lo que permite a cualquier programa que conozca el formato leerlo correctamente.

Las clases **DataOutputStream** y **DataInputStream** de java.io sirven para leer y escribir ficheros binarios estructurados.

Métodos de DataOutputStream	Descripción
<code>writeln(int)</code>	Escribe un entero con signo. Entero (4 bytes)
<code>writeDouble(double)</code>	Escribe un número en coma flotante. Decimal (8 bytes)
<code>writeFloat(float)</code>	Escribe un número float. Decimal (4 bytes)

writeLong(long)	Escribe un long. Entero largo (8 bytes)
writeBoolean(boolean)	Escribe un valor verdadero/falso. Booleano (1 byte)
writeChar(char)	Escribe un carácter Unicode. Carácter (2 bytes)
writeUTF(String)	Escribe una cadena precedida por su longitud en 2 bytes. Cadena UTF-8
writeByte(int)	Escribe un solo byte. Byte (1 byte)
writeShort(int)	Escribe un short. Entero corto (2 bytes)

Métodos de DataInputStream	Descripción
readInt()	Lee un entero con signo (Entero)
readDouble()	Lee un número double (Decimal)
readFloat()	Lee un número float (Decimal)
readLong()	Lee un long (Entero largo)
readBoolean()	Lee un valor verdadero/falso (Booleano)
readChar()	Lee un carácter Unicode (Carácter)
readUTF()	Lee una cadena UTF-8 (Cadena UTF-8)
readByte()	Lee un byte (Byte)
readShort()	Lee un short (Entero corto)

**Ejemplo 10:** Lectura y escritura en ficheros binarios estructurados (con tipos primitivos):

```
import java.io.DataInputStream
import java.io.DataOutputStream
import java.io.FileInputStream
import java.io.FileOutputStream
import java.nio.file.Files
import java.nio.file.Path

fun main() {
    val ruta = Path.of("multimedia/binario.dat")
```

```

Files.createDirectories(ruta.parent)

// Escritura binaria
val fos = FileOutputStream(ruta.toFile())
val out = DataOutputStream(fos)
out.writeInt(42)           // int (4 bytes)
out.writeDouble(3.1416)    // double (8 bytes)
out.writeUTF("K")         // char (2 bytes)
out.close()
fos.close()
println("Fichero binario escrito con DataOutputStream.")

// Lectura binaria
val fis = FileInputStream(ruta.toFile())
val input = DataInputStream(fis)
val entero = input.readInt()
val decimal = input.readDouble()
val caracter = input.readUTF()
input.close()
fis.close()

println("Contenido leído:")
println("  Int: $entero")
println("  Double: $decimal")
println("  Char: $caracter")
}

```



Ejecuta el ejemplo anterior y comprueba que la salida es la siguiente:

```

Fichero binario escrito con DataOutputStream.
Contenido leído:
Int: 42
Double: 3.1416
Char: K

```

## 5.3. Ficheros de imágenes

Las imágenes son ficheros binarios que contienen datos que representan gráficamente una imagen visual (fotografías, ilustraciones, etc.). A diferencia de los ficheros de texto o binarios crudos, un fichero de imagen tiene estructura interna que depende del formato. Algunos de los más comunes son:

- **.jpg**: Comprimido con pérdida, ideal para fotos
- **.png**: Comprimido sin pérdida, soporta transparencia
- **.bmp**: Sin compresión, ocupa más espacio
- **.gif**: Admite animaciones simples, limitada a 256 colores

Método	Descripción
ImageIO.read(Path/File) ImageIO.write(BufferedImage, ...)	Usa javax.imageio.ImageIO

### Ejemplo 11: Generación de una imagen

```

import java.awt.Color
import java.awt.image.BufferedImage
import java.io.File
import javax.imageio.ImageIO

fun main() {
    val ancho = 200
    val alto = 100
    val imagen = BufferedImage(ancho, alto, BufferedImage.TYPE_INT_RGB)
    // Rellenar la imagen con colores
    for (x in 0 until ancho) {
        for (y in 0 until alto) {
            val rojo = (x * 255) / ancho
            val verde = (y * 255) / alto
            val azul = 128
            val color = Color(rojo, verde, azul)
            imagen.setRGB(x, y, color.rgb)
        }
    }
    // Guardar la imagen
    val archivo = File("multimedia/imagen_generada.png")
    ImageIO.write(imagen, "png", archivo)
    println("Imagen generada correctamente: ${archivo.getAbsolutePath}")
}

```



Ejecuta el ejemplo anterior y verifica que se crea la imagen correctamente.

### Ejemplo 12: Conversión de una imagen a escala de grises

```

import java.awt.Color
import java.awt.image.BufferedImage
import java.nio.file.Files
import java.nio.file.Path
import java.nio.file.StandardCopyOption
import javax.imageio.ImageIO

fun main() {

```

```

val originalPath = Path.of("multimedia/jpg/amanecer1.jpg")
val copiaPath = Path.of("multimedia/jpg/amanecer1_copia.jpg")
val grisPath = Path.of("multimedia/jpg/amanecer1_escala_de_grises.png")

// 1. Comprobar si la imagen existe
if (!Files.exists(originalPath)) {
    println("No se encuentra la imagen original: $originalPath")
} else {
    // 2. Copiar la imagen con java.nio (para no modificar el original)
    Files.copy(originalPath, copiaPath, StandardCopyOption.REPLACE_EXISTING)
    println("Imagen copiada a: $copiaPath")

    // 3. Leer la imagen en un objeto BufferedImage
    val imagen: BufferedImage = ImageIO.read(copiaPath.toFile())

    // 4. Convertir a escala de grises, píxel por píxel
    for (x in 0 until imagen.width) {
        for (y in 0 until imagen.height) {
            // Obtenemos el color del píxel actual.
            val color = Color(imagen.getRGB(x, y))

            /* Calcular el valor de gris usando la fórmula de Luminosidad.
               Esta fórmula pondera los colores rojo, verde y azul según la
               sensibilidad del ojo humano. El resultado es un único valor de brillo que
               convertimos a entero. */
            val gris = (color.red * 0.299 + color.green * 0.587 + color.blue * 0.114).toInt()

            // Creamos un nuevo color donde los componentes rojo, verde y azul
            // son todos iguales al valor de 'gris' que hemos calculado.
            val colorGris = Color(gris, gris, gris)

            // Establecemos el nuevo color gris en el píxel de la imagen.
            imagen.setRGB(x, y, colorGris.rgb)
        }
    }

    // 5. Guardar la imagen modificada. Usamos "png" porque es un formato sin
    // pérdida, ideal para imágenes generadas.
    ImageIO.write(imagen, "png", grisPath.toFile())
    println("Imagen convertida a escala de grises y guardada como: $grisPath")
}
}

```

 Ejecuta el ejemplo anterior y verifica que la imagen generada es la misma que la original pero en tonos de grises.

## 6. Ficheros binarios de acceso aleatorio

Un fichero binario de acceso aleatorio es un tipo de fichero que permite leer o escribir en cualquier posición del fichero directamente, sin necesidad de procesar secuencialmente todo el contenido previo. El sistema puede “saltar” a una posición concreta (medida en bytes desde el inicio del fichero) y comenzar la lectura o escritura desde ahí. Por ejemplo, si cada registro ocupa 200 bytes, para acceder al registro número 100 hay que saltar  $200 \times 99 = 19.800$  bytes desde el inicio.

Las clases **FileChannel**, **ByteBuffer** y **StandardOpenOption** se utilizan juntas para leer y escribir en ficheros binarios y en el acceso aleatorio a ficheros. **ByteBuffer** se utiliza en ficheros de acceso aleatorio porque permite leer y escribir bloques binarios de datos en posiciones específicas del fichero.

Métodos de FileChannel	Descripción
position()	Devuelve la posición actual del puntero en el fichero y permite saltar a cualquier posición en él (tanto para leer como para escribir).
position(long)	Establece una posición exacta para lectura/escritura
truncate(long)	Recorta o amplía el tamaño del fichero
size()	Devuelve el tamaño total actual del fichero
read(ByteBuffer) write(ByteBuffer)	Usa FileChannel para secuencial o aleatorio

Métodos de ByteBuffer	Descripción
allocate(capacidad)	Crea un buffer con capacidad fija en memoria (no compartida).
wrap(byteArray)	Crea un buffer que envuelve un array de bytes existente (memoria compartida).
wrap(byteArray, offset, length)	Crea un buffer desde una porción del array existente.
put(byte)	Escribe un byte en la posición actual.
.putInt(int)	Escribe un valor int.
putDouble(double)	Escribe un valor double.

putFloat(float)	Escribe un valor float.
putChar(char)	Escribe un carácter (char, 2 bytes).
putShort(short)	Escribe un valor short.
putLong(long)	Escribe un valor long.
put(byte[], offset, length)	Escribe una porción de un array de bytes
get()	Lee un byte desde la posición actual.
getInt()	Lee un valor int.
getDouble()	Lee un valor double.
getFloat()	Lee un valor float.
getChar()	Lee un carácter (char).
getShort()	Lee un valor short.
getLong()	Lee un valor long.
get(byte[], offset, length)	Lee una porción del buffer a un array.

Métodos de control del buffer	Descripción
position()	Devuelve la posición actual del cursor.
position(int)	Establece la posición del cursor.
limit()	Devuelve el límite del buffer.
limit(int)	Establece un nuevo límite.
capacity()	Devuelve la capacidad total del buffer.
clear()	Limpia el buffer: posición a 0, límite al máximo (sin borrar contenido).
flip()	Prepara el buffer para lectura después de escribir.
rewind()	Posición a 0 para releer desde el inicio.
remaining	Indica cuántos elementos quedan por procesar.

hasRemaining()	true si aún queda contenido por leer o escribir.
----------------	--

Antes de seguir, hay que dejar claro los tipos de fichero de texto (algunos de ellos ya los hemos visto en los puntos anteriores).

Extensión	Contenido típico	Comentario didáctico
.txt	Texto plano	Puede abrirse en cualquier editor de texto. Fácil de leer y modificar manualmente.
.csv	Texto plano, valores separados por comas	Muy usado para tablas, hojas de cálculo y bases de datos ligeras.
.dat	Binario o texto genérico	“Archivo de datos genérico”. Se usaba mucho en ejemplos educativos y programas antiguos. No indica claramente si es texto o binario.
.bin	Binario puro	Fichero con información directamente en bytes. Más moderno y descriptivo para datos binarios. Ideal para mostrar cómo se almacena información en memoria.

**IMPORTANTE:** un fichero .bin o un fichero .dat binario **no es un fichero de texto plano**. No se puede abrir con el Bloc de Notas,TextEdit, o un editor de código en modo texto normal ya que se ven caracteres extraños, símbolos y espacios.

**Ejemplo 13:** El siguiente ejemplo utiliza FileChannel y ByteBuffer para crear y leer un fichero binario con información sobre plantas en el que cada registro de planta tendrá estos campos, con longitud fija:

Campo	Tipo	Tamaño fijo
id_planta	Int	4 bytes
nombre_comun	String	20 bytes
altura_maxima	Double	8 bytes

```
import java.nio.ByteBuffer
import java.nio.channels.FileChannel
import java.nio.charset.Charset
import java.nio.file.Files
```

```

import java.nio.file.Path
import java.nio.file.Paths
import java.nio.file.StandardCopyOption
import java.nio.file.StandardOpenOption
data class PlantaBinaria(val id_planta: Int, val nombre_comun: String, val
altura_maxima: Double)

// Tamaño fijo para cada registro en el fichero
const val TAMANO_ID = Int.SIZE_BYTES           // 4 bytes
const val TAMANO_NOMBRE = 20                   // String de tamaño fijo 20 bytes
const val TAMANO_ALTURA = Double.SIZE_BYTES    // 8 bytes
const val TAMANO_REGISTRO = TAMANO_ID + TAMANO_NOMBRE + TAMANO_ALTURA

//Función que crea un fichero (si no existe) o lo vacía (si existe)
//Si el fichero existe: CREATE se ignora. TRUNCATE_EXISTING se activa y
vacía el fichero en el momento de abrirlo.
//Si el fichero no existe: CREATE se activa y crea un fichero nuevo y
vacío. TRUNCATE_EXISTING se ignora porque el fichero no existía
previamente a la operación open.
fun vaciarCrearFichero(path: Path) {
    try {
        // AÑADIMOS StandardOpenOption.CREATE
        // WRITE: Permite la escritura.
        // CREATE: Crea el fichero si no existe.
        // TRUNCATE_EXISTING: Si el fichero ya existía, lo vacía. Si fue
recién creado, esta opción se ignora.
        FileChannel.open(path, StandardOpenOption.WRITE,
StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING).close()
        println("El fichero '${path.fileName}' existe y está vacío.")
    } catch (e: Exception) {
        println("Error al vaciar o crear el fichero: ${e.message}")
    }
}

fun anadirPlanta(path: Path, idPlanta: Int, nombre: String, altura: Double) {
    val nuevaPlanta = PlantaBinaria(idPlanta, nombre, altura)

    // Abrimos el canal en modo APPEND.
    // CREATE: crea el fichero si no existe.
    // WRITE: es necesario para poder escribir.
    // APPEND: asegura que cada escritura se haga al final del fichero.
    try {
        FileChannel.open(path, StandardOpenOption.WRITE,
StandardOpenOption.CREATE, StandardOpenOption.APPEND).use { canal ->
        // Creamos un buffer para la nueva planta
        val buffer = ByteBuffer.allocate(TAMANO_REGISTRO)
        // Llenamos el buffer con los datos de la nueva planta
        // (misma lógica que en la función de escritura inicial)
    }
}

```

```

        buffer.putInt(nuevaPlanta.id_planta)
    val nombreBytes = nuevaPlanta.nombre_comun
        .padEnd(20, ' ')
        .toByteArray(Charset.defaultCharset())
    buffer.put(nombreBytes, 0, 20)
    buffer.putDouble(nuevaPlanta.altura_maxima)
    // Preparamos el buffer para ser leído por el canal
    buffer.flip()
    // Escribimos el buffer en el canal. Gracias a APPEND,
    // se escribirá al final del fichero.
    while (buffer.hasRemaining()) {
        canal.write(buffer)
    }
    println("Planta '${nuevaPlanta.nombre_comun}' añadida con éxito.")
}
} catch (e: Exception) {
    println("Error al añadir la planta: ${e.message}")
}
}

fun leerPlantas(path: Path): List<PlantaBinaria> {
    val plantas = mutableListOf<PlantaBinaria>()

    // Abrimos un canal de solo Lectura al fichero
    FileChannel.open(path, StandardOpenOption.READ).use { canal ->
        // Buffer para Leer un registro a la vez
        val buffer = ByteBuffer.allocate(TAMANO_REGISTRO)
        // Leer del canal al buffer hasta el final del fichero (-1)
        while (canal.read(buffer) > 0) {
            // Preparamos el buffer para la lectura de datos
            buffer.flip()
            // Leemos los datos en el mismo orden en que se escribieron
            val id = buffer.getInt()
            val nombreBytes = ByteArray(TAMANO_NOMBRE)
            buffer.get(nombreBytes)
                val nombre = String(nombreBytes,
            Charset.defaultCharset()).trim()
            val altura = buffer.getDouble()
            plantas.add(PlantaBinaria(id, nombre, altura))
            // Compactamos o limpiamos el buffer para la siguiente lectura
            buffer.clear()
        }
    }
    return plantas
}

fun main() {
    val archivoPath: Path = Paths.get("plantas.bin")
}

```

```

val lista = listOf(
    PlantaBinaria(1, "Rosa", 1.5),
    PlantaBinaria(2, "Girasol", 3.0),
    PlantaBinaria(3, "Margarita", 0.6)
)

// vaciar o crear el fichero
vaciarCrearFichero(archivoPath)

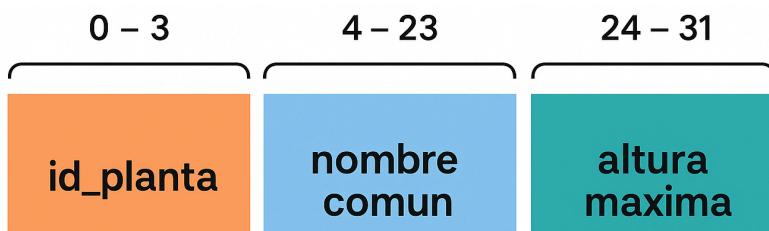
// --- Añadir las plantas al fichero
lista.forEach { planta ->
    anadirPlanta(archivoPath, planta.id_planta, planta.nombre_comun,
planta.altura_maxima)
}
// --- Leer el fichero binario ---
val leidas = leerPlantas(archivoPath)
println("Plantas leídas del fichero:")
for (dato in leidas) {
    println("      - ID: ${dato.id_planta}, Nombre común:
${dato.nombre_comun}, Altura: ${dato.altura_maxima} metros")
}
}

```

El fichero .bin se podrá abrir (para ver el contenido "crudo") con un **editor hexadecimal** como por ejemplo:

- **HxD** en Windows <https://mh-nexus.de/en/hxd/>
- **xxd** en Linux
- **<https://hexed.it/>** (Editor online)

Cada registro ocupará exactamente **32 bytes**.



El fichero contendrá registros de 32 bytes que irán seguidos sin separadores:

Offset	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	ASCII
00000000	00 00 00 01 52 6F 73 61 20 20 20 20 20 20 20 20	....Rosa
00000010	20 20 20 20 20 20 3F F8 00 00 00 00 00 00 00 00	?.....
00000020	00 02 47 69 72 61 73 6F 6C 20 20 20 20 20 20 20 20	..Girasol
00000030	20 20 20 20 20 40 08 00 00 00 00 00 00 00 00 00	....@.....
00000040	03 4D 61 72 67 61 72 69 74 61 20 20 20 20 20 20	.Margarita
00000050	20 20 20 20 3F E3 33 33 33 33 33 33 33 33 33 33	....?.333333

Equivale a la siguiente información:

### Registro 1: PlantaBinaria(1, "Rosa", 1.5)

- **ID (1):** 00 00 00 01 (en hexadecimal)
- **Nombre ("Rosa"):** 52 6F 73 61 20 20 20 20 20 20 20 20 20 20 20 20
  - 52 -> 'R', 6F -> 'o', 73 -> 's', 61 -> 'a'
  - 20 -> '' (espacio, repetido 16 veces para llenar hasta 20 bytes)
- **Altura (1.5):** 3F F8 00 00 00 00 00 00 (representación estándar IEEE 754 para un double)

### Registro 2: PlantaBinaria(2, "Girasol", 3.0)

- **ID (2):** 00 00 00 02
- **Nombre ("Girasol"):** 47 69 72 61 73 6F 6C 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
  - 47->'G', 69->'i', 72->'r', 61->'a', 73->'s', 6F->'o', 6C->'l'
  - 20 -> '' (espacio, repetido 13 veces)
- **Altura (3.0):** 40 08 00 00 00 00 00 00

### Registro 3: PlantaBinaria(3, "Margarita", 0.6)

- **ID (3):** 00 00 00 03
- **Nombre ("Margarita"):** 4D 61 72 67 61 72 69 74 61 20 20 20 20 20 20 20 20 20 20 20
  - 4D->'M', 61->'a', 72->'r', 67->'g', 61->'a', 72->'r', 69->'i', 74->'t', 61->'a'
  - 20 -> '' (espacio, repetido 11 veces)
- **Altura (0.6):** 3F E3 33 33 33 33 33 33



Ejecuta el ejemplo anterior y comprueba que la salida es la siguiente:

El fichero 'plantas.bin' existe y está vacío.  
 Planta 'Rosa' añadida con éxito.  
 Planta 'Girasol' añadida con éxito.  
 Planta 'Margarita' añadida con éxito.

Plantas leídas del fichero:

- ID: 1, Nombre común: Rosa, Altura: 1.5 metros
- ID: 2, Nombre común: Girasol, Altura: 3.0 metros
- ID: 3, Nombre común: Margarita, Altura: 0.6 metros

•

**Ejemplo 14:** Ahora que ya tenemos la información guardada en nuestro fichero .bin y sabemos leerla, vamos a ampliar la aplicación con una nueva función: modificar la altura de una planta. El siguiente código recoge la ruta del fichero, el id de la planta a modificar y su nueva altura y, cuando localiza el registro de la planta cuyo ID coincide con el que buscamos, escribe la nueva información en la posición correspondiente.

```
fun modificarAlturaPlanta(path: Path, idPlanta: Int, nuevaAltura: Double)
{
    // Abrimos un canal con permisos de Lectura y escritura
    FileChannel.open(path, StandardOpenOption.READ,
StandardOpenOption.WRITE).use { canal ->
    val buffer = ByteBuffer.allocate(TAMANO_REGISTRO)
    var encontrado = false
    while (canal.read(buffer) > 0 && !encontrado) {

        /*canal.read(buffer) Lee un bloque completo de 32 bytes
        y los guarda en el buffer. Después de esta operación,
        el "puntero" o "cursor" interno del canal (canal.position())
        ha avanzado 32 bytes y ahora se encuentra al final
        del registro que acabamos de leer.*/
        val posicionActual = canal.position()
        buffer.flip()

        val id = buffer.getInt()
        if (id == idPlanta) {
            // Hemos encontrado el registro. Calculamos la posición del
            campo 'altura'.
            // Posición de inicio del registro (actual - registro) + 4
            bytes (id) + 20 bytes (nombre)
            val posicionAltura = posicionActual - TAMANO_REGISTRO +
TAMANO_ID + TAMANO_NOMBRE

            // Creamos un buffer solo para el double
            val bufferAltura = ByteBuffer.allocate(TAMANO_ALTURA)
            bufferAltura.putDouble(nuevaAltura)
    }
}
```

```
        bufferAltura.flip()

        // Escribimos el nuevo valor en la posición exacta del
fichero
        canal.write(bufferAltura, posicionAltura)
        encontrado = true
    }
    buffer.clear()
}
if (encontrado) {
    println("Altura de la planta con ID $idPlanta modificada a
$nuevaAltura")
} else {
    println("No se encontró la planta con ID $idPlanta")
}
}
```

La llamada en el main sería:

```
// --- Modificar una planta
modificarAlturaPlanta(archivoPath, 2, 5.5)
// --- Volver a Leer para verificar el cambio ---
val leidasDespuesDeModificar = LeerPlantas(archivoPath)
println("Plantas leídas después de la modificación:")
for (dato in leidasDespuesDeModificar) {
    println(" - ID: ${dato.id_planta}, Nombre común: ${dato.nombre_comun},
Altura: ${dato.altura_maxima} metros")
}
```



Realiza los siguientes pasos:

1. Añade al código del ejemplo anterior la función `modificarAlturaPlanta()`
  2. Añade al main las llamadas a `modificarAlturaPlanta()` y a `leerPlantas()`.
  3. Ejecuta la aplicación y comprueba que la salida es la siguiente:

El fichero 'plantas.bin' existe y está vacío.

Planta 'Rosa' añadida con éxito.

Planta 'Girasol' añadida con éxito.

Planta 'Margarita' añadida con éxito.

#### **Plantas leídas del fichero:**

- ID: 1. Nombre común: Rosa. Altura: 1.5 metros

- ID: 2. Nombre común: Girasol. Altura: 3.0 metros

- ID: 3. Nombre común: Margarita. Altura: 0.6 metros

Altura de la planta con ID 2 modificada a 5.5

#### Plantas leídas después de la modificación:

- ID: 1, Nombre común: Rosa, Altura: 1.5 metros
- ID: 2, Nombre común: Girasol, Altura: 5.5 metros
- ID: 3, Nombre común: Margarita, Altura: 0.6 metros

**Ejemplo 15:** Por último, ampliaremos la aplicación para poder eliminar una planta a partir de su ID. El programa recorre todo los registros comprobando si el ID coincide con el buscado. En caso de que no coincida escribe el registro en un fichero temporal y si coincide, no hace nada. Al finalizar el fichero temporal contendrá los registros que no se quieren eliminar. Por último, se elimina el fichero original y se renombra el fichero temporal con el nombre original.

```
fun eliminarPlanta(path: Path, idPlanta: Int) {
    // Creamos un fichero temporal en el mismo directorio
    val pathTemporal = Paths.get(path.toString() + ".tmp")
    var plantaEncontrada = false

    // Abrimos el canal de lectura para el fichero original
    FileChannel.open(path, StandardOpenOption.READ).use { canalLectura ->
        // Abrimos el canal de escritura para el fichero temporal
        FileChannel.open(pathTemporal, StandardOpenOption.WRITE,
        StandardOpenOption.CREATE).use { canalEscritura ->
            val buffer = ByteBuffer.allocate(TAMANO_REGISTRO)
            // Leemos el fichero original registro a registro
            while (canalLectura.read(buffer) > 0) {
                buffer.flip()
                val id = buffer.getInt() // Solo necesitamos el ID
                if (id == idPlanta) {
                    plantaEncontrada = true
                    // Si es la planta a eliminar, no hacemos nada.
                    // El buffer se limpiará para la siguiente lectura.
                } else {
                    // Si NO es la planta a eliminar, escribimos el registro
                    // completo en el fichero temporal.
                    buffer.rewind() // Rebobinamos para ir al principio
                    canalEscritura.write(buffer)
                }
                buffer.clear() // Preparamos para siguiente iteración
            }
        }
    }

    if (plantaEncontrada) {
```

```
// Si se encontró y eliminó, reemplazar fichero original con el temporal
    Files.move(pathTemporal, path, StandardCopyOption.REPLACE_EXISTING)
    println("Planta con ID $idPlanta eliminada con éxito.")
} else {
    // Si no se encontró, no hace falta hacer nada, borramos el temporal
    Files.delete(pathTemporal)
    println("No se encontró ninguna planta con ID $idPlanta.")
}
}
```

La llamada en el main sería:

```
// --- Eliminar una planta
eliminarPlanta(archivoPath, 3)
// --- Volver a leer para verificar el cambio ---
val leidasDespuesDeEliminar = LeerPlantas(archivoPath)
println("Plantas leídas después de la modificación:")
for (dato in leidasDespuesDeEliminar) {
    println(" - ID: ${dato.id_planta}, Nombre común: ${dato.nombre_comun},  
Altura: ${dato.altura_maxima} metros")
}
```



Realiza los siguientes pasos:

1. Añade el código de la función `eliminarPlanta()` al ejemplo anterior.
2. Añade al main la llamada a `eliminarPlanta()`.
3. Ejecuta la aplicación y comprueba que la salida es la siguiente:

*El fichero 'plantas.bin' existe y está vacío.*  
*Planta 'Rosa' añadida con éxito.*  
*Planta 'Girasol' añadida con éxito.*  
*Planta 'Margarita' añadida con éxito.*  
*Plantas leídas del fichero:*  
*- ID: 1, Nombre común: Rosa, Altura: 1.5 metros*  
*- ID: 2, Nombre común: Girasol, Altura: 3.0 metros*  
*- ID: 3, Nombre común: Margarita, Altura: 0.6 metros*  
*Altura de la planta con ID 2 modificada a 5.5*  
*Plantas leídas después de la modificación:*  
*- ID: 1, Nombre común: Rosa, Altura: 1.5 metros*  
*- ID: 2, Nombre común: Girasol, Altura: 5.5 metros*  
*- ID: 3, Nombre común: Margarita, Altura: 0.6 metros*  
*Planta con ID 3 eliminada con éxito.*  
*Plantas leídas después de la modificación:*

- ID: 1, Nombre común: Rosa, Altura: 1.5 metros
- ID: 2, Nombre común: Girasol, Altura: 5.5 metros

## 🎯 Práctica 7: Gestión de ficheros binarios

Realiza lo siguiente:

1. **Diseña tu registro binario:** A partir de tu data class, define la estructura del registro de tamaño fijo que se guardará en el fichero .bin. Calcula el tamaño total del registro en bytes. (Int = 4 bytes, Double = 8 bytes, para los String, asigna una longitud fija en bytes y gestiona cómo llenar el espacio sobrante (espacios o bytes nulos) al escribir, y cómo limpiarlo (.trim()) al leer).
2. **Crea la función vaciarCrearFichero()** que debe vaciar o crear tu fichero binario (*datos\_finales.bin*) dentro de la carpeta *datos\_fin*.
3. **Crea la función anadir()** que tomará los datos de un nuevo elemento y añadirá un nuevo registro al final de tu fichero .bin (usando StandardOpenOption.APPEND)
4. **Crea la función importar():** que leerá la información de tu fichero (CSV, XML o JSON) y los escribirá (uno por uno) llamando a la función **anadir()** en tu fichero binario.
5. **Crea la función mostrar():** Debe abrir el fichero .bin, leerlo de forma secuencial y mostrar toda la información por consola de forma ordenada.
6. **Comprueba:** Desde el main llama a las funciones anteriores para verificar que el ciclo de escritura y lectura funciona.
7. **Crea la función modificar():** Pedirá por consola el id del registro a modificar y buscará ese registro en el fichero. Si lo encuentra, pedirá los nuevos datos (mínimo dos). Para saltar a la posición exacta de ese registro utilizará acceso aleatorio (FileChannel.position()) y sobrescribirá únicamente los campos a modificar, sin alterar el resto del fichero.
8. **Crea la función eliminar():** Pedirá por consola el id del registro a modificar. Implementa la técnica vista en el ejemplo, (leer el fichero original registro a registro, escribir los que se conservan en un fichero temporal, borrar el original y renombrar el temporal).
9. **Comprueba:** Prueba estas funciones desde main, llamando a **mostrar()** antes y después de cada operación para verificar los resultados.

### Aspectos Técnicos Obligatorios:

- La gestión del fichero .bin debe realizarse obligatoriamente con las clases *java.nio.file.Path*, *java.nio.channels.FileChannel* y *java.nio.ByteBuffer*.

## 7. Documentación: El Fichero LEEME.md

En un proyecto de software, el código fuente por sí solo no cuenta toda la historia, es fundamental crear documentación adicional. La forma estándar hacerlo es a través de un fichero LEEME.md (o README.md) ubicado en la raíz del proyecto, es decir, al mismo nivel que build.gradle.kts, settings.gradle.kts y la carpeta src/.

El fichero LEEME.md es lo primero que verá cualquier persona (incluido nuestro "yo" del futuro) que quiera entender nuestro código. Es buena práctica explicar qué hace el proyecto, cómo se utiliza y por qué se tomaron algunas decisiones, por ejemplo ¿por qué elegimos un registro de 36 bytes?" o "¿por qué el nombre del fichero es registros.dat?". Un buen fichero LEEME.md debería contener, como mínimo, las siguientes secciones:

- **Nombre del proyecto y breve descripción.**
  - **Estructura de Datos:** En esta sección se explica el diseño de los datos.
  - **Instrucciones de Ejecución:** Pasos claros y sencillos para que otra persona pueda ejecutar nuestro programa.
  - **Decisiones de Diseño** (Opcional pero Recomendado): Un pequeño apartado para explicar brevemente por qué tomamos ciertas decisiones.

La extensión .md significa **Markdown** que es un lenguaje de marcado ligero que permite dar formato a un texto plano usando caracteres simples. Podemos crearlo con cualquier editor de texto (IntelliJ, VSCode, Bloc de notas...) y guardarla con la extensión .md. Plataformas como GitHub, GitLab y otros sistemas de documentación convierten estos ficheros en páginas web.

## Sintaxis básica de Markdown para empezar

```
# Título de Nivel 1
## Título de Nivel 2
### Título de Nivel 3
**Texto en negrita**
*Texto en cursiva*
- Elemento de una lista
1. Elemento de una lista numerad
Para bloques de código, rodearlos con tres comillas invertidas
(```) y especificar el lenguaje:
```kotlin
fun main() {
    println("Hola, Markdown!")
}

```

### Ejemplo 16: Ejemplo de fichero LEEME.md:

```
# Gestor de plantas
```

Este es un programa de consola desarrollado en Kotlin para gestionar una colección de plantas. Los datos se almacenan en un fichero binario de acceso aleatorio llamado \*plantas.bin\*

```
## 1. Estructura de datos
```

```
### **Data Class:**
```

```
```kotlin
```

```
data class Sensor(  
    val id_planta: Int,  
    val nombre_comun: String,  
    val altura_maxima: Double  
)  
```
```

```
### **Estructura del registro binario:**
```

- **id\_planta**: Int - 4 bytes
- **nombre\_comun**: String - 20 bytes (longitud fija)
- **altura\_maxima**: Double - 8 bytes
- **Tamaño Total del Registro**: 4 + 20 + 8 = 32 bytes

```
## 2. Instrucciones de ejecución
```

- **Requisitos previos**: Asegúrate de tener un JDK (ej. versión 17 o superior) instalado.
- **Compilación**: Abre el proyecto en IntelliJ IDEA y deja que Gradle sincronice las dependencias.
- **Ejecución**: Ejecuta la función main del fichero Main.kt.
- **Ficheros necesarios**: El programa espera encontrar un fichero **\*plantas.csv\*** en la carpeta **\*datos\_ini\*** dentro de la raíz del proyecto para la carga inicial de datos.

```
## 3. Decisiones de diseño
```

- Elegí CSV para los datos iniciales porque es un formato muy fácil de crear y editar manualmente con cualquier hoja de cálculo.
- Decidí que el campo nombre tuviera 20 bytes porque considero que es suficiente para la mayoría de nombres de plantas sin desperdiciar demasiado espacio.

## 🎯 Práctica 8: El Producto Final

Realiza lo siguiente:

1. **Crea el menú principal:** Con las siguientes opciones:

1. Mostrar todos los registros
2. Añadir un nuevo registro
3. Modificar un registro (por ID)
4. Eliminar un registro (por ID)
5. Salir

El menú se **mantendrá en ejecución** hasta que el usuario decida salir y:

- **Validará la entrada del usuario:** El programa no se detendrá si el usuario introduce texto en lugar de un número. Usa `toIntOrNull()` y `try-catch` donde sea necesario.
- Cada **opción del menú** llamará a una función:
  - **Mostrar todos:** Llamará a la función `mostrarTodo()`
  - **Añadir registro:** Pedirá al usuario los datos del nuevo registro y llamará a `nuevoReg()`
  - **Modificar registro:** Pedirá al usuario el ID del registro a modificar y llamará a la función `modificar()`
  - **Eliminar registro:** Pedirá al usuario el ID del registro a eliminar y llamará a la función `eliminar()`
- 2. **Escribe la documentación:** Crea el fichero LEEME.md para tu proyecto. Explica tu temática, la estructura de datos que diseñaste (la data class y el registro binario incluyendo su tamaño total) y las instrucciones para ejecutar tu programa.

**¡Enhорabuena, has completado tu proyecto!**



### Entrega final

Entrega el código fuente del proyecto comprimido en un fichero .zip para que la profesora lo califique (el programa entregado deberá ejecutarse, si da error de ejecución, no se corregirá).

Para calificar tu trabajo se utilizará la guía de calificación del ANEXO 1.

## ANEXO 1: Guía de calificación

EL PROGRAMA ENTREGADO DEBERÁ EJECUTARSE (si da error de ejecución, no se corregirá). Para calificar se tendrá en cuenta lo siguiente:

| Apartados                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | Ptos |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| <ul style="list-style-type: none"> <li>- El proyecto está configurado correctamente con <b>Gradle</b>.</li> <li>- El <b>código está bien estructurado</b> con una única función main que lanza el menú principal (es funcional, navega entre opciones y "Salir" funciona).</li> <li>- Todas las funciones son legibles y tienen nombres claros.</li> <li>- Se utilizan las clases java.nio.Path y java.nio.file.Files para gestionar rutas.</li> <li>- Se gestionan adecuadamente las excepciones de E/S (IOException, FileNotFoundException) y de entrada de usuario (NumberFormatException). La aplicación no se detiene inesperadamente.</li> <li>- El fichero LEEME.md describe la estructura del registro binario y las instrucciones de ejecución de la aplicación en lenguaje MarkDown.</li> </ul> | 2    |
| Se define una <b>data class</b> para representar un registro de información y se <b>lee</b> correctamente el fichero de origen elegido.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | 1    |
| Se <b>crea</b> (si no existe) o se <b>vacía</b> el fichero .bin y se <b>escribe</b> en él respetando estrictamente la estructura de registro de tamaño fijo (tipos, tamaños y relleno de Strings) con java.nio.channels.FileChannel y java.nio.ByteBuffer.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | 3    |
| Se lee y muestra correctamente los datos del fichero .bin de forma secuencial.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | 1    |
| Se localiza el registro por ID y se sobrescribe <b>solo</b> los campos requeridos sin corromper el fichero binario utilizando <b>acceso aleatorio</b> de forma eficiente con FileChannel.position().                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      | 3    |