

Acceso a Datos

2º DAM

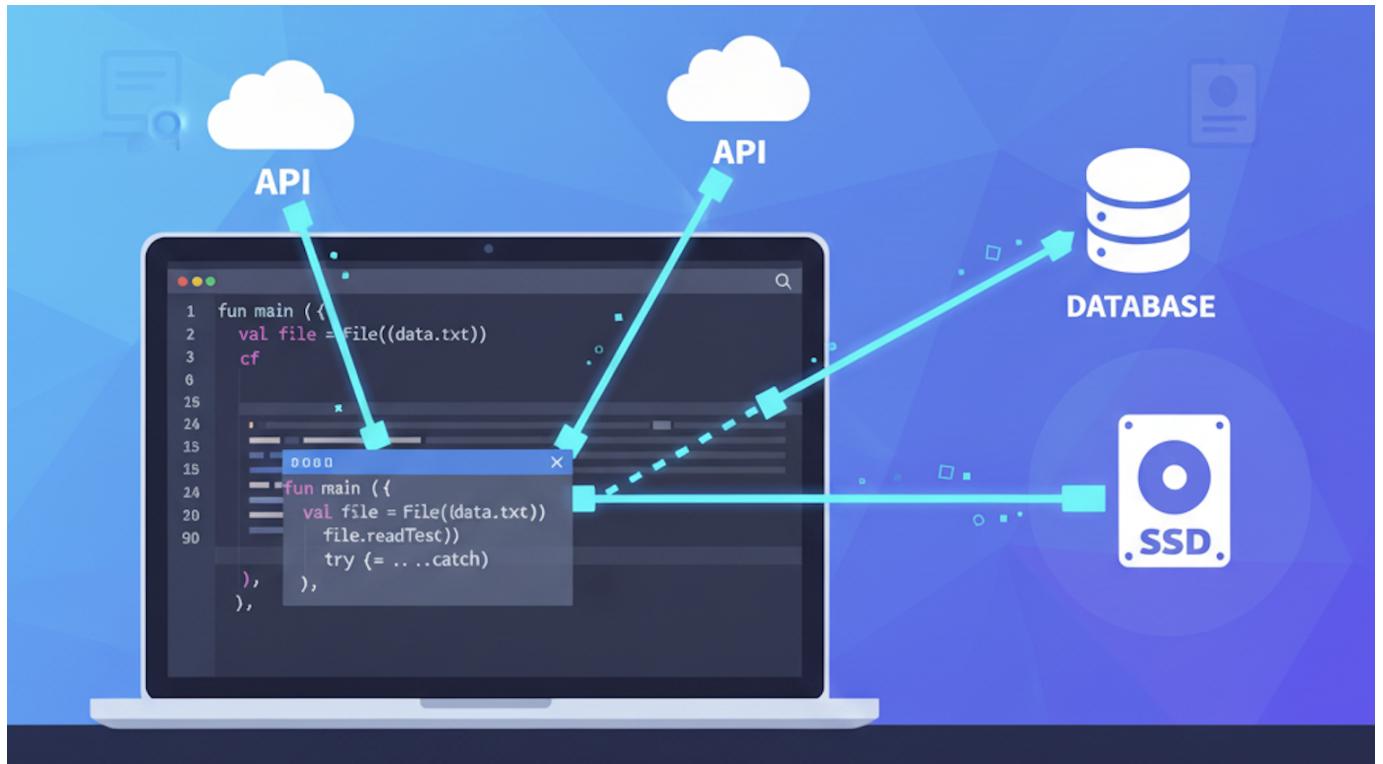
José Manuel Abad López

© Creative Commons CC BY-SA 4.0 2025

Contenidos

1. Acceso a datos DAM	3
1.1 Descripción	3
1.2 Enlaces de interés	3
1.3 Autoría y revisión	3
1.4 Repositorio y contacto	3
2. Unidades	4
2.1 UD 1 Introducción a Kotlin	4
2.1.1 1. Introducción	4
2.1.2 2. Instalación de IntelliJ Community	4
2.1.3 3. Proyectos	5
2.1.4 4. Variables	9
2.1.5 5. Entrada y salida estándar	11
2.1.6 6. Condicionales	12
2.1.7 7. Repeticiones	13
2.1.8 8. Estructuras de datos	14
2.1.9 9. Funciones	15
2.1.10 10. POO	17
2.2 UD2 - Persistencia en ficheros	25
2.2.1 UD2 - Persistencia en ficheros	25
2.2.2 Ficheros y directorios	28
2.2.3 Ficheros de texto	33
2.2.4 Ficheros de intercambio	35
2.2.5 Ficheros binarios	43
2.2.6 Ficheros de acceso aleatorio	47
2.2.7 Documentación final	53

1. Acceso a datos DAM



1.1 Descripción

Material del módulo de **Acceso a Datos de 2º de DAM** del IES CAMP DE MORVEDRE.

Fecha	Versión	Descripción
19/09/2025	1.0.0	Adaptación del material. Estructura web con mkdocs .

1.2 Enlaces de interés

- Todo el material está disponible en el [repositorio del módulo](#)

1.3 Autoría y revisión

Obra realizada por **Begoña Paterna** basado en materiales de **Alicia Salvador**. Adaptado, revisado y ampliado por **José Manuel Abad López**.

Publicada bajo licencia Creative Commons Atribución/Reconocimiento-CompartirIgual 4.0 Internacional

1.4 Repositorio y contacto

<https://jmabadlopez.github.io> jmabadlopez@edu.gva.es

2. Unidades

2.1 UD 1 Introducción a Kotlin

En este documento se realiza una introducción al lenguaje de programación Kotlin utilizando el IDE IntelliJ en su versión Community.



2.1.1 1. Introducción

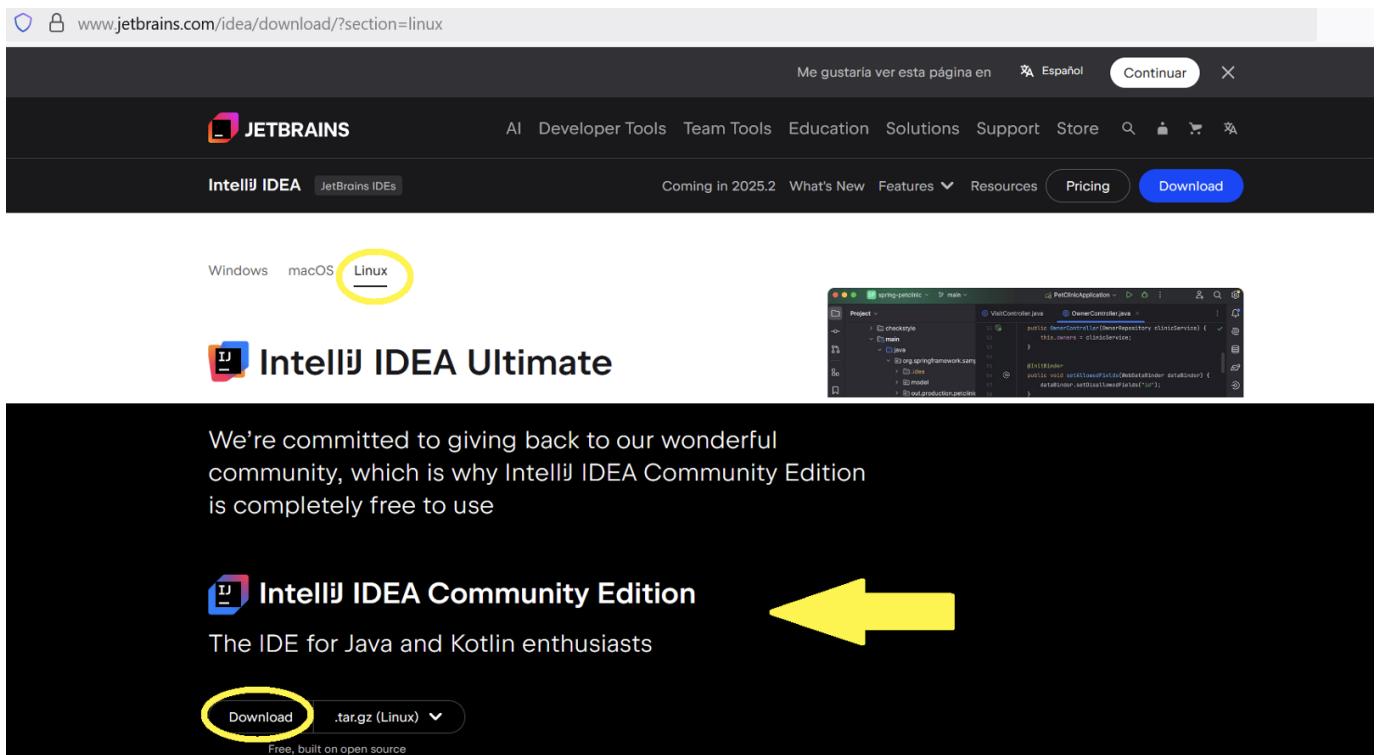
En el módulo de Programación, de 1º de DAM, hemos trabajado con Java, un lenguaje orientado a objetos y ampliamente utilizado en el desarrollo de aplicaciones empresariales. Aprendimos conceptos como estructuras de control, clases, objetos y herencia.

Este curso, en el módulo de Acceso a Datos, vamos a continuar aplicando estos mismos conceptos utilizando Kotlin, un lenguaje moderno, conciso y seguro. Kotlin funciona sobre la máquina virtual de Java (JVM), fue desarrollado por JetBrains (los creadores de IntelliJ IDEA) y es 100% interoperable con Java.

Los principales usos de Kotlin son: aplicaciones Android, aplicaciones de escritorio (con Swing, JavaFX o Compose Desktop), backends web (con frameworks como Ktor o Spring) y desarrollo multiplataforma (Kotlin Multiplatform).

2.1.2 2. Instalación de IntelliJ Community

En este curso utilizaremos el IDE IntelliJ en su versión Community. Para instalarlo podemos descargarlo del enlace oficial <https://www.jetbrains.com/idea/download/>. Dependiendo de nuestro sistema operativo, escogeremos el instalador que mejor se ajuste a nuestras necesidades.



2.1.3.3. Proyectos

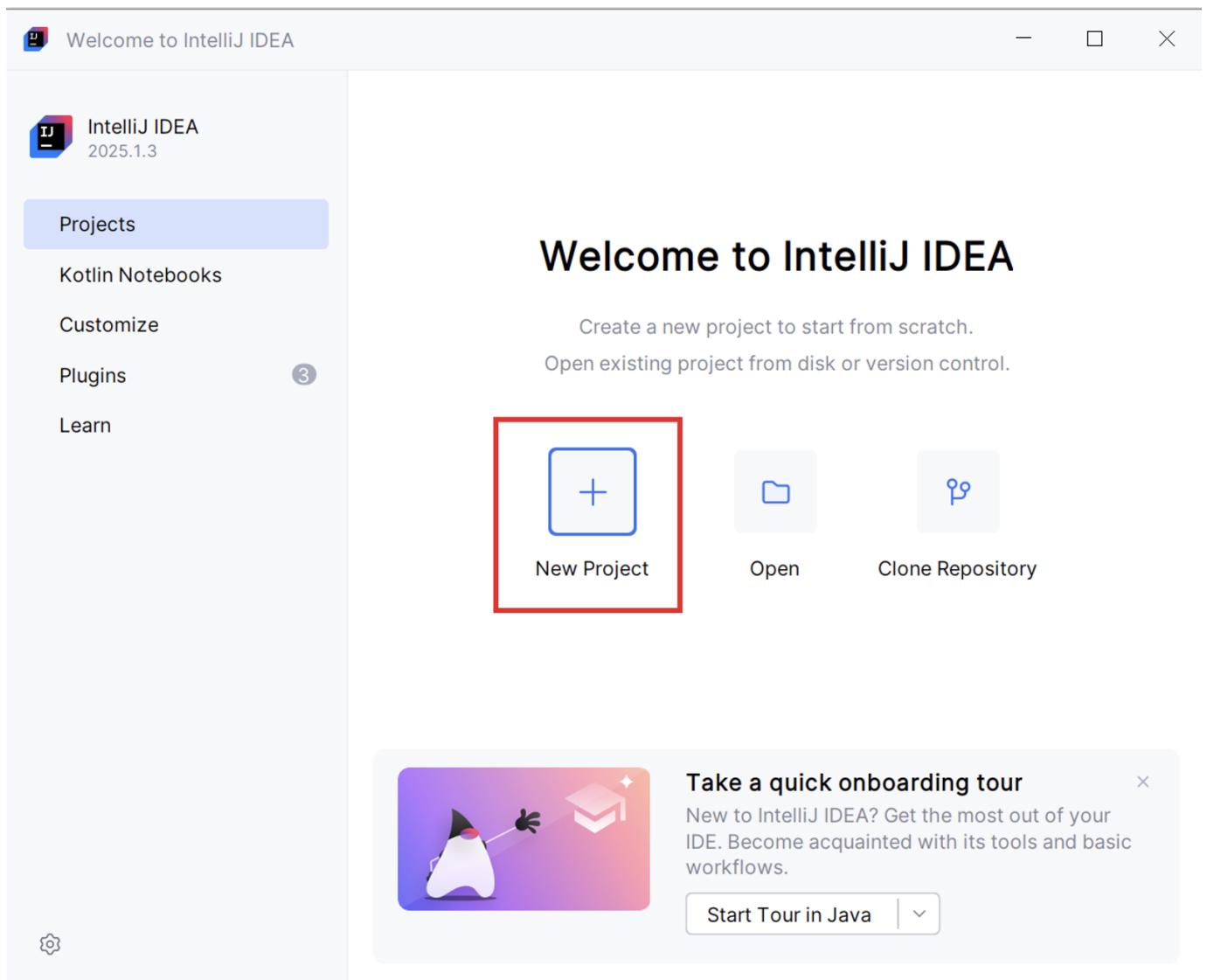
Antes de empezar hay que tener en cuenta cómo vamos a organizar nuestros proyectos. Cada uno de nosotros podemos tener una metodología de trabajo distinta (puede que unos prefieran crear un proyecto para cada ejercicio, otros un proyecto por unidad, etc). En los ejemplos de estos materiales, se ha creado una carpeta de trabajo llamada `kot` en la que se guardarán los distintos proyectos.

También podemos configurar el entorno de trabajo a nuestro gusto entrando al menú de ajustes. Para ello hacemos clic en el icono de la rueda dentada que hay en la esquina inferior izquierda de la pantalla principal del programa.

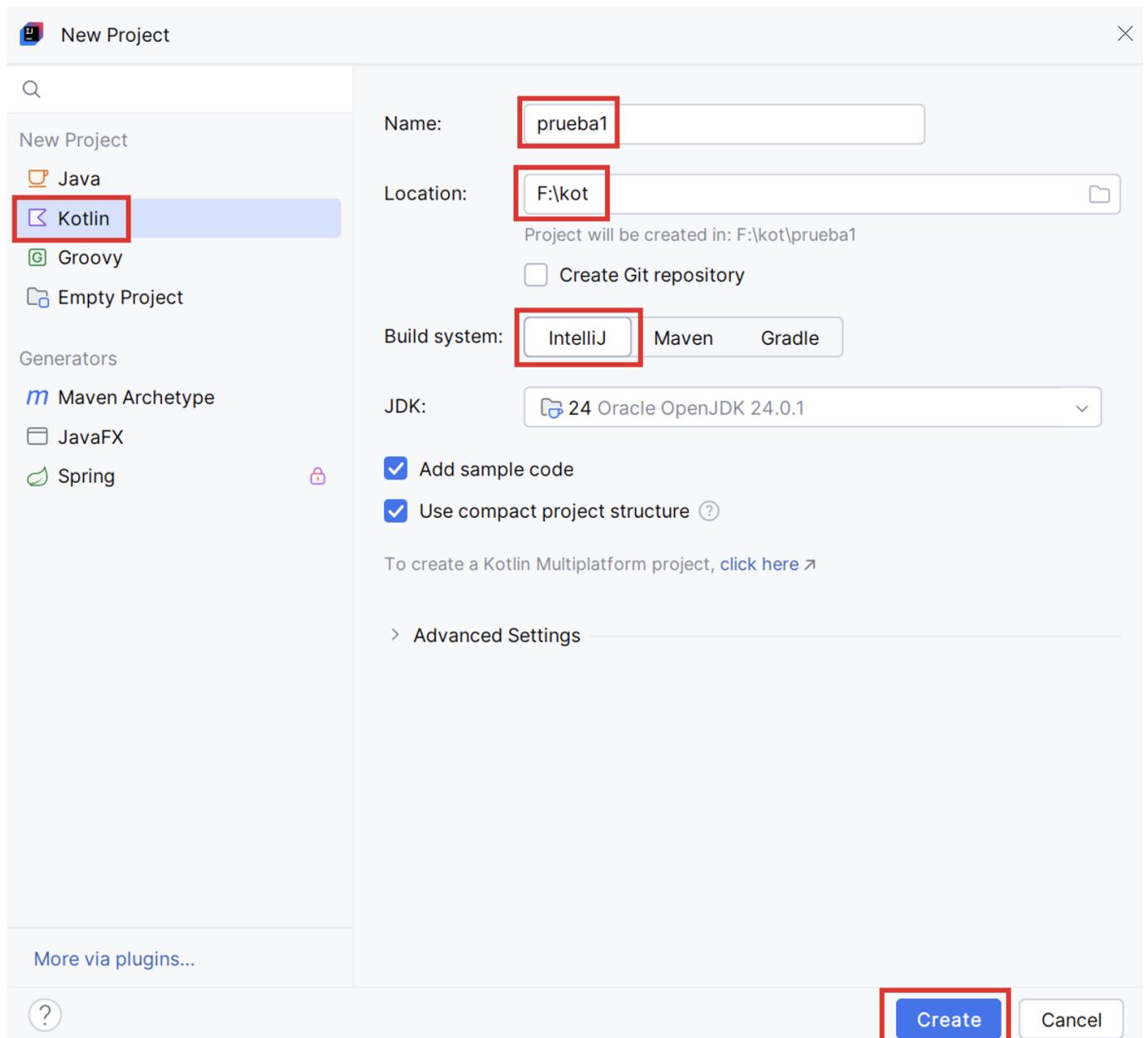
Una vez realizadas estas aclaraciones, ya podemos trabajar con proyectos. A continuación veremos algunos aspectos importantes.

3.1. Creación del proyecto

Para crear nuestro proyecto debemos realizar los siguientes pasos: - Abrir la aplicación y, en la ventana inicial, hacer clic en el ícono **New Project**.



- Indicar los parámetros del proyecto:
 - Marcar la opción **Kotlin** en la columna izquierda.
 - Indicar el nombre del proyecto y su ubicación.
 - Asegurarnos que aparece seleccionado el sistema **IntelliJ**.



- Por último, hacer clic en el botón **Create**.

Con estos pasos ya tendremos nuestro nuevo proyecto (`prueba1`) que aparecerá en una nueva ventana con un pequeño programa de ejemplo.

```

    fun main() {
        val name = "Kotlin"
        println("Hello, " + name + "!")
        for (i in 1 .. 5) {
            println("i = $i")
        }
    }

```

3.2. Ejecución de la aplicación

Para ejecutarla hay que hacer clic en el ícono **Play**. El proyecto debe tener, al menos, un método `main`, Kotlin utiliza la palabra reservada `fun` para declararlo. El resultado de la ejecución aparecerá por consola (en la parte inferior de la pantalla).

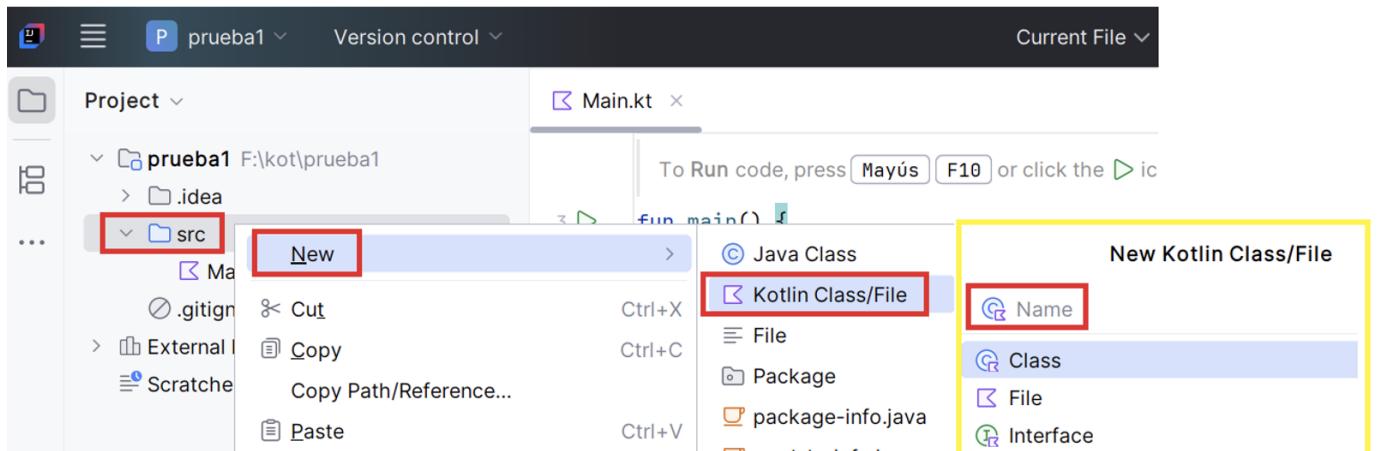
```

Hello, Kotlin!
i = 1
i = 2
i = 3
i = 4
i = 5

```

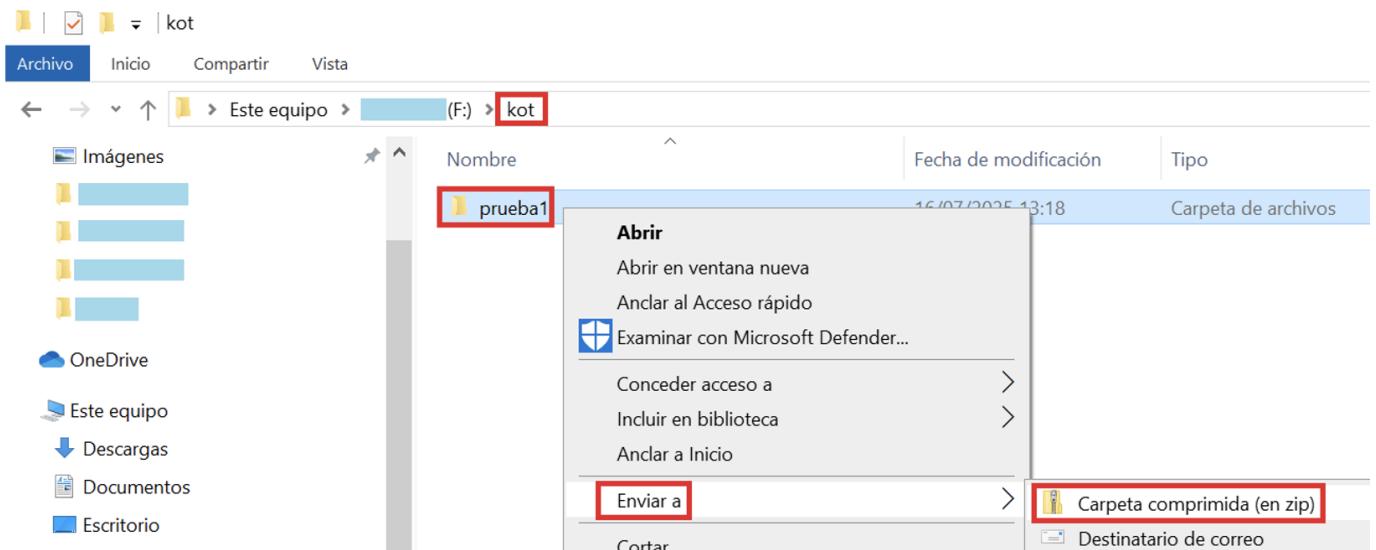
3.3. Estructura de carpetas

Como vimos en Java, al crear un nuevo proyecto se crea una estructura de carpetas y los archivos fuentes deben estar en la carpeta `src`. En este caso, Kotlin funciona exactamente igual. Para crear una nueva clase o archivo hay que hacer clic con el botón derecho del ratón sobre carpeta `src`, luego hacer clic en **New**, luego clic en **Kotlin File/Class** y por último indicar el nombre.



3.4. Compartir proyectos

Por último vamos a recordar cómo compartir proyectos (para entregar en una tarea de clase, para hacer copia de seguridad, etc). Se realiza desde fuera de IntelliJ utilizando el explorador de archivos del sistema operativo. Para ello hay que localizar la carpeta correspondiente al proyecto y comprimirla.



2.1.4 4. Variables

Una **variable** es un espacio en memoria que guarda un dato (un número, un texto, etc). En Kotlin existen dos formas principales de declarar variables:

4.1. val – Inmutable (no se puede cambiar)

Es como una constante: una vez que le das un valor, no puedes cambiarlo.

```
val pi = 3.1416
println ("El valor de PI es $pi")
```

4.2. var – Mutable (se puede cambiar)

Es una variable normal: puedes cambiar su valor más adelante.

```
var contador = 0
contador += 1
println ("Contador: $contador") // Salida: Contador: 1
```

4.3. Tipos de datos

En Kotlin no es necesario declarar el tipo de una variable (aunque puede hacerse). A continuación se detallan los tipos más comunes:

Tipo	Descripción	Ejemplo
Byte	Entero pequeño (8 bits)	<code>val a: Byte = 1</code>
Short	Entero corto (16 bits)	<code>val b: Short = 100</code>
Int	Entero estándar (32 bits)	<code>val c: Int = 1000</code>
Long	Entero largo (64 bits). Se debe añadir la L al final del número	<code>val d: Long = 100000L</code>
Float	Decimal de precisión simple (32 bits). Se debe añadir la f al final del número	<code>val e: Float = 1.5f</code>
Double	Decimal de precisión doble (64 bits). Es el tipo por defecto	<code>val f: Double = 3.14</code>
Char	Carácter individual	<code>val letra: Char = 'A'</code>
String	Texto (También se puede acceder a los caracteres individualmente)	<code>val saludo: String = "Hola Mundo" val letra = saludo[0] // 'H'</code>
Boolean	Verdadero o falso	<code>val esMayor = true</code>

A continuación se muestran los operadores aritméticos:

Operador	Nombre	Ejemplo	Resultado
+	Suma	<code>5 + 3</code>	8
-	Resta	<code>10 - 4</code>	6
*	Multiplicación	<code>6 * 2</code>	12
/	División (entera o real)	<code>9 / 3</code>	3
%	Módulo (resto de una división)	<code>10 % 3</code>	1

En Kotlin, una división entre enteros da como resultado un número entero: `val resultado = 7 / 2 // Resultado: 3`

Para obtener decimales, al menos uno debe ser tipo Double o Float: `val resultado = 7.0 / 2 // Resultado: 3.5`

En Kotlin, Por defecto, las variables no pueden ser nulas, el siguiente código daría un error de compilación ya que nombre no puede ser null:

```
var nombre: String = "Pol"
nombre = null // Esto dará un error de compilación
```

Para permitir valores nulos, se usa el operador `?`. Ahora el ejemplo ya es correcto:

```
var nombre: String? = "Pol"
nombre = null
```

Kotlin tiene unos operadores para evitar errores cuando se trabaja con variables que pueden ser null. A continuación se describen estos operadores:

Operador	Nombre	Descripción
<code>?.</code>	Llamada segura	Ejecuta el método solo si la variable no es null
<code>?:</code>	Elvis operator	Devuelve un valor por defecto si la variable es null.
<code>!!</code>	Not null assertion	Fuerza el acceso a la variable. Si es null, lanza <code>NullPointerException</code> .

Ejemplo 1 - Llamada segura (? .):

```
var nombre: String? = null
println (nombre?.length) // No lanza error, devuelve: null
nombre = "Pol"
println (nombre?.length) // Devuelve 3
```

Ejemplo 2 - Valor por defecto (?:):

```
var nombre: String? = null
var mostrar = nombre ?: "Desconocido"
println (mostrar) // Si nombre es null imprime: Desconocido
nombre = "Pol"
mostrar = nombre ?: "Desconocido"
println (mostrar) // Si nombre es "Pol" imprime: Pol
```

Ejemplo 3 - Acceso forzado (!!):

```
val nombre: String? = null
println (nombre!!.length) // Lanza NullPointerException
```

Ejemplo 4 - Uso con if:

```
if (nombre != null) {
    println ("Hola, ${nombre.uppercase()}")
} else {
    println ("Nombre no disponible")
}
```

2.1.5 5. Entrada y salida estándar

Cuando queremos mostrar información al usuario o pedirle información utilizamos la consola. Las funciones más comunes para comunicarnos con él son:

Función	Descripción	Ejemplo	Salida
print()	Imprime texto en pantalla, sin salto de línea al final.	print("Hola ") print("mundo")	Hola mundo
println()	Imprime texto en pantalla y añade un salto de línea al final.	println("Hola") println("mundo")	Hola mundo
readLine()	Lee una línea de texto que el usuario escribe por teclado.		

readLine() devuelve un valor de tipo String (que puede ser nulo). Hay que convertirlo si se espera otro tipo de dato (toInt(), toDouble(), etc.). Antes de la conversión hay que asegurarse de que no está vacío.

```
fun main() {
    print ("Introduce tu edad: ")
    val entrada = readLine()
    if (entrada != null && entrada.isNotBlank()) {
        val edad = entrada.toInt()
        println ("Tendrás ${edad + 1} años el próximo año")
    } else {
        println ("Edad no válida")
    }
}
```

2.1.6 6. Condicionales

Las condiciones en Kotlin se tratan de manera muy parecida a Java y para ello se utilizan los operadores relacionales.

Operador	Significado	Ejemplo	Resultado
<code>==</code>	Igual a	<code>5 == 5</code>	<code>true</code>
<code>!=</code>	Distinto de	<code>5 != 3</code>	<code>true</code>
<code>></code>	Mayor que	<code>10 > 7</code>	<code>true</code>
<code><</code>	Menor que	<code>3 < 8</code>	<code>true</code>
<code>>=</code>	Mayor o igual que	<code>6 >= 6</code>	<code>true</code>
<code><=</code>	Menor o igual que	<code>4 <= 9</code>	<code>true</code>

Estos operadores pueden combinarse con los operadores lógicos siguientes:

Operador	Nombre	Ejemplo	Resultado
<code>&&</code>	AND	<code>(edad > 18 && tieneID)</code>	<code>true si ambas son true</code>
<code> </code>	OR	<code>(edad > 18 tieneID)</code>	<code>true si alguna es true</code>
<code>!</code>	NOT	<code>!esActivo</code>	Invierte el valor: <code>true → false</code>

Ejemplo 1 - condicional simple:

```
val edad = 18
if (edad >= 18) {
    println ("Eres mayor de edad")
}
```

Ejemplo 2 - if-else:

```
val numero = 5
if (numero % 2 == 0) {
    println ("Es par")
} else {
    println ("Es impar")
}
```

Ejemplo 3 - if anidados:

```
val edad = 22
val tieneID = true
val esEmpleado = false
val tienePaseEspecial = true
if (edad >= 18 && tieneID) {
    println ("Edad y documento verificados")
    if (esEmpleado || tienePaseEspecial) {
        println ("Acceso permitido a la zona restringida")
    } else {
        println ("Acceso denegado: no eres empleado o no tienes pase")
    }
} else {
    println ("Acceso denegado: no cumples con edad o documentación")
}
```

Ejemplo 4 - como expresión (devuelve un valor):

```
val max = if (a > b) a else b
```

Ejemplo 5 - when (como switch):

```
val dia = 3
val nombreDia = when (dia) {
    1 -> "Lunes"
    2 -> "Martes"
    3 -> "Miércoles"
```

```

    else -> "Día inválido"
}

```

Ejemplo 6 - when con condiciones:

```

val nota = 85
val resultado = when {
    nota >= 90 -> "Excelente"
    nota >= 70 -> "Aprobado"
    else -> "Reprobado"
}

```

2.1.7 7. Repeticiones

A continuación se ven las estructuras repetitivas `while`, `do-while`, `for` y `repeat`.

7.1. while

Repite mientras la condición sea verdadera. Se evalúa la condición **antes** de entrar al ciclo. Si la condición es falsa desde el principio, el bloque **no se ejecuta**.

```

var contador = 1
while (contador <= 5) {
    println ("Contador: $contador")
    contador++
}

```

7.2. do-while

Hace la acción al menos una vez, luego verifica la condición. Se ejecuta **primero el bloque de código**, y **luego** se evalúa la condición.

```

var contador = 1
do {
    println ("Contador: $contador")
    contador++
} while (contador <= 5)

```

7.3. for

Recorre un rango, lista o secuencia. Se usa cuando **sabemos cuántas veces** queremos repetir algo.

```

for (i in 1..5) {
    println ("Iteración: $i")
}

```

7.4. repeat

Repite una acción N veces (sin necesidad de un rango ni una colección).

```

repeat(3) {
    println("Hola")
}

```

Estructura	¿Cuándo usarla?	¿Evalúa primero?	¿Se ejecuta al menos una vez?
<code>while</code>	Cuando no sabemos cuántas veces , pero depende de una condición.	Sí	No
<code>do-while</code>	Cuando queremos que se ejecute al menos una vez .	No	Sí
<code>for</code>	Cuando sabemos cuántas veces o queremos recorrer algo.	Sí	Sí
<code>repeat</code>	Cuando queremos repetir algo un número fijo de veces.	-	-

2.1.8 8. Estructuras de datos

Las estructuras de datos son formas de organizar, almacenar y manipular información de manera eficiente. Las principales estructuras de datos en Kotlin son:

8.1. Array

Colección ordenada de tamaño fijo. Todos sus elementos son del mismo tipo. A cada elemento se accede mediante un índice. Se utiliza cuando se tiene un número fijo de elementos del mismo tipo.

- 1. Crear un array vacío con un tamaño fijo y luego llenarlo:

```
val calificaciones = IntArray(5) // Los 5 enteros con valor 0
calificaciones[0] = 95
```

- 2. Asignar los valores directamente:

```
val edades = intArrayOf(15, 18, 20)
```

- 3. Utilizar arrayOf :

```
val numeros = arrayOf(10, 20, 30, 40, 50)
```

No existe una clase concreta para manejar el tipo String y por tanto se utiliza el método genérico `arrayOf` visto en el ejemplo anterior:

```
val nombres = arrayOf("Pol", "Eli", "Ade")
println(nombres[1]) // Salida: Eli
```

- Acceder a los elementos de un array:

```
println(numeros[0]) // Salida: 10
println(numeros[3]) // Salida: 40
```

- Cambiar un valor del array:

```
numeros[2] = 99
println(numeros[2]) // Salida: 99
```

- Tamaño del array:

```
println("Tamaño del array: ${numeros.size}")
```

- 2 formas de recorrer un array con un for:

```
for (numero in numeros) {
    println(numero)
}
for (i in 0..numeros.size - 1) {
    println(numeros[i])
}
```

8.2. List (lista)

Colección ordenada que permite elementos duplicados. Puede ser inmutable (`List`) o mutable (`MutableList`). Se utiliza cuando se quiere mantener un orden y permitir elementos repetidos.

```
val nombres = listOf("Pol", "Eli", "Ade") // No modificable
println(nombres) // [Pol, Eli, Ade]

val nombresMutable = mutableListOf("Pol", "Eli")
nombresMutable.add("Ade") // Añadir un elemento al final
nombresMutable.add(1, "Fer") // Añadir en una posición específica
nombresMutable[1] = "Sam" // Cambiar el valor en una posición específica
nombresMutable.remove("Sam") // Eliminar por valor
nombresMutable.removeAt(0) // Eliminar por índice
nombresMutable.clear() // Eliminar todos los elementos
```

8.3. Set (conjunto)

Colección sin duplicados, sin orden garantizado. Puede ser inmutable (`Set`) o mutable (`MutableSet`). Se utiliza cuando no se quieren duplicados y el orden no importa.

```
val set = setOf("Rojo", "Verde", "Rojo") // "Rojo" solo se guarda una vez
println(set) // Imprime: [Rojo, Verde]

val frutas = mutableSetOf("Manzana", "Banana", "Uva")
frutas.add("Naranja") // Agregar un elemento
println(frutas)
frutas.add("Banana") // Intentar agregar un duplicado
println(frutas)
frutas.remove("Uva") // Eliminar un elemento
println(frutas)

if ("Banana" in frutas) {
    println("Sí hay Banana")
}
for (fruta in frutas) {
    println("Fruta: $fruta")
}
```

8.4. Map (diccionario o mapa)

Colección de pares clave → valor. Cada clave es única; útil para representar relaciones. Puede ser inmutable (`Map`) o mutable (`MutableMap`). Se utiliza cuando se quiere asociar claves con valores.

```
val edades = mapOf("Pol" to 18, "Ade" to 20)
println(edades["Ade"]) // Imprime 20

val datos = mutableMapOf<String, Int>()
datos["Pol"] = 25
datos["Eli"] = 20
println("Pol:" + datos["Pol"])
println("Eli:" + datos["Eli"])
```

Estructura	Ordenada	Claves únicas	Permite duplicados	Modificable
Array	Sí	No	Sí	(depende)
List	Sí	No	Sí	si es mutable
Set	No	Sí	No	si es mutable
Map	No	Sí (en claves)	en valores	si es mutable

2.1.9 9. Funciones

Son bloques de código que realizan tareas específicas (métodos en Java) y sirven para organizar, reutilizar y evitar repetir el mismo código varias veces.

- **Ejemplo de función sin parámetros ni retorno:**

```
fun saludar() {
    println("¡Hola, bienvenidos a la clase!")
}
saludar()
```

- **Ejemplo de función con parámetros:**

```
fun saludarEstudiante(nombre: String) {
    println("Hola, $nombre")
}
saludarEstudiante("Pol")
```

- **Ejemplo de función que devuelve un valor (número entero):**

```
fun sumar(a: Int, b: Int): Int {
    return a + b
}
```

```
}
```

• Ejemplo de función con forma simplificada:

```
fun multiplicar(a: Int, b: Int) = a * b
var multi = multiplicar(3, 5)
```

• Ejemplo de función con parámetro con valor por defecto:

```
fun saludar(nombre: String = "Pol") {
    println("Hola, $nombre")
}
saludar()
saludar("Eli")
```

• Ejemplo de función con parámetros combinados (algunos con valores por defecto):

```
fun mostrarMensaje(mensaje: String, veces: Int = 1) {
    repeat(veces) {
        println(mensaje)
    }
}
mostrarMensaje("¡Hola!")
mostrarMensaje("¡Hola!", 3)
```

• Ejemplo de llamada con argumentos nombrados:

```
fun mostrarMensaje(mensaje: String, veces: Int = 1) {
    repeat(veces) {
        println(mensaje)
    }
}
mostrarMensaje("Hola a todos", 2)
mostrarMensaje(mensaje = "Hola a todos", veces = 2)
mostrarMensaje(veces = 2, mensaje = "Hola a todos")
```

• Ejemplo de función con un array de enteros como parámetro y retorno:

```
fun duplicarValores(numeros: Array<Int>): Array<Int> {
    val resultado = Array(numeros.size) { i -> numeros[i] * 2 }
    return resultado
}
val original = arrayOf(1, 2, 3)
val duplicados = duplicarValores(original)
println("Original: ${original.joinToString()}")
println("Duplicados: ${duplicados.joinToString()}")
```

• Ejemplo de función con un array de Strings como parámetro y retorno:

```
fun agregarSigno(nombres: Array<String>): Array<String> {
    return Array(nombres.size) { i -> "${nombres[i]}!" }
}
val nombres = arrayOf("Pol", "Eli", "Ade")
val nombresConSigno = agregarSigno(nombres)
println("Original: ${nombres.joinToString()}")
println("Con signo: ${nombresConSigno.joinToString()}")
```

9.1. Funciones locales

Una **función local** es una función que se define **dentro de otra función**. Solo puede ser usada dentro de esa función de forma interna.

```
fun procesarTexto(texto: String) {
    fun limpiar(cadena: String): String {
        return cadena.trim().lowercase()
    }
    val resultado = limpiar(texto)
    println("Texto procesado: $resultado")
}
procesarTexto(" Hola Mundo ")
```

9.2. Funciones con cantidad variable de argumentos

Un parámetro de una función puede recibir una cantidad variable de argumentos (0 o más), como si fuera un "array flexible" pero sin necesidad de pasarlo como un array. Para ello se utiliza la palabra clave `vararg`.

```
fun saludarVarios(vararg nombres: String) {
    for (nombre in nombres) {
        println("Hola, $nombre")
    }
}
saludarVarios("Pol", "Eli")
```

- **Ejemplo de combinar vararg con otros parámetros:**

```
fun mostrarNumeros(titulo: String, vararg numeros: Int) {
    println(titulo)
    for (n in numeros) {
        println(n)
    }
}
mostrarNumeros("Lista de números:", 3, 5, 7)
```

- **Ejemplo pasando valores desde un array:**

```
fun imprimirNumeros(vararg numeros: Int) {
    for (n in numeros) {
        println(n)
    }
}
val lista = intArrayOf(1, 2, 3, 4)
imprimirNumeros(*lista)

val extra = intArrayOf(4, 5)
imprimirNumeros(1, 2, 3, *extra, 6)
```

9.3. Funciones de orden superior

Una **función de orden superior** es una función que **trabaja con funciones como si fueran datos**.

- **Ejemplo 1:**

```
fun opera(a: Int, b: Int, op: (Int, Int) -> Int): Int {
    return op(a, b)
}
val suma = opera(3, 4) { x, y -> x + y }
val resta = opera(10, 5) { x, y -> x - y }
println("Suma: $suma")
println("Resta: $resta")
```

- **Ejemplo 2:**

```
fun crearMultiplicador(factor: Int): (Int) -> Int {
    return { numero -> numero * factor }
}
val porTres = crearMultiplicador(3)
println(porTres(5))
```

2.1.10 10. POO

La **Programación Orientada a Objetos (POO)** es una forma de escribir programas donde todo gira en torno a **objetos**. Un **objeto** es una combinación de **datos** (como características o propiedades) y **métodos** (acciones que puede hacer). En POO, usamos **clases** para crear estos objetos.

Ejemplo 1 - Constructor primario:

```
class Estudiante(val nombre: String, val edad: Int) {
    // Método para imprimir los datos del estudiante
    fun imprimirDatos() {
        println("Nombre: $nombre, Edad: $edad")
    }
    // Método para verificar si es menor de edad
    fun esMenorDeEdad(): Boolean {
        return edad < 18
    }
}
fun main() {
```

```

    val estudiante1 = Estudiante("Pol", 16)
    estudiante1.imprimirDatos()
    println("¿Es menor de edad? ${estudiante1.esMenorDeEdad()}\n")
    val estudiante2 = Estudiante("Eli", 20)
    estudiante2.imprimirDatos()
    println("¿Es menor de edad? ${estudiante2.esMenorDeEdad()}")
}

```

Ejemplo 2 - Propiedades dentro del cuerpo:

```

class Estudiante() {
    var nombre: String = ""
    var edad: Int = 0
    fun imprimirDatos() {
        println("Nombre: $nombre, Edad: $edad")
    }
    fun esMenorDeEdad(): Boolean {
        return edad < 18
    }
}
fun main() {
    val estudiante1 = Estudiante()
    estudiante1.nombre = "Pol"
    estudiante1.edad = 16
    estudiante1.imprimirDatos()
    println("¿Es menor de edad? ${estudiante1.esMenorDeEdad()}\n")
    val estudiante2 = Estudiante()
    estudiante2.nombre = "Eli"
    estudiante2.edad = 20
    estudiante2.imprimirDatos()
    println("¿Es menor de edad? ${estudiante2.esMenorDeEdad()}")
}

```

Ejemplo 3 - constructor secundario:

```

class Estudiante {
    var nombre: String = ""
    var edad: Int = 0
    // Constructor secundario
    constructor(nombre: String, edad: Int) {
        this.nombre = nombre
        this.edad = edad
    }
    fun imprimirDatos() {
        println("Nombre: $nombre, Edad: $edad")
    }
    fun esMenorDeEdad(): Boolean {
        return edad < 18
    }
}

```

10.1. Constructor primario + secundario

```

class Estudiante(val nom: String, val edad: Int, val direccion: String) {
    // Constructor secundario que asigna una dirección vacía
    constructor(nom: String, edad: Int) : this(nom, edad, "Sin dirección")
    fun imprimirDatos() {
        println("Nombre: $nom, Edad: $edad, Dirección: $direccion")
    }
    fun esMenorDeEdad(): Boolean {
        return edad < 18
    }
}
fun main() {
    val estudiante1 = Estudiante("Pol", 16, "Calle Mayor")
    estudiante1.imprimirDatos()
    println("¿Es menor de edad? ${estudiante1.esMenorDeEdad()}\n")
    val estudiante2 = Estudiante("Eli", 20)
    estudiante2.imprimirDatos()
    println("¿Es menor de edad? ${estudiante2.esMenorDeEdad()}")
}

```

10.2. getters y setters

`get` y `set` son mecanismos para **acceder y modificar propiedades**, y forman parte del **encapsulamiento y control de acceso**. En Kotlin **todas las propiedades (var)** tienen automáticamente un **getter y un setter**.

- **Ejemplo 1 - Getters y Setters automáticos:**

```
class Estudiante {
    var nombre: String = "Sin nombre"
}
fun main() {
    val estudiante = Estudiante()
    estudiante.nombre = "Pol"      // setter automático
    println(estudiante.nombre)    // getter automático. Salida: Pol
}
```

- **Ejemplo 2 - Personalizar el getter y el setter:**

```
class Producto {
    var nombre: String = "Sin nombre"
    var precio: Double = 0.0
        get() {
            println("Obteniendo precio...")
            return field
        }
        set(value) {
            println("Asignando precio: $value")
            field = if (value >= 0) value else 0.0
        }
}
fun main() {
    val producto = Producto()
    producto.precio = 120.0
    println("Precio actual: ${producto.precio}")
    producto.precio = -50.0
    println("Precio actual: ${producto.precio}")
    producto.nombre = "Teclado inalámbrico"
    println("Nombre del producto: ${producto.nombre}")
}
```

- **Ejemplo 3 - Propiedad de solo lectura (val con get personalizado):**

```
class Circulo(val radio: Double) {
    val area: Double
        get() = Math.PI * radio * radio
}
fun main() {
    val c = Circulo(5.0)
    println("Área del círculo: ${c.area}")
}
```

- **Ejemplo 4 - Setter privado:**

```
class Usuario(val nombre: String) {
    var edad: Int = 0
        private set // Solo se puede modificar dentro de la clase
    fun cumpleaños() {
        edad++
    }
}
fun main() {
    val usuario = Usuario("Pol")
    usuario.cumpleaños()
    println("Edad: ${usuario.edad}")
    // usuario.edad = 30 // Esto daría un error de compilación
}
```

10.3. Relación entre clases

Normalmente en una aplicación necesitaremos programar varias clases que se relacionarán unas con otras.

```
class Curso(val nombre: String, val duracionSemanas: Int) {
    fun infoCurso(): String {
        return "Curso: $nombre, Duración: $duracionSemanas semanas"
    }
}
class Estudiante(
    val nombre: String,
    val edad: Int,
    val dirección: String,
    val curso: Curso
```

```

) {
    constructor() : this("Sin nombre", 0, "Sin direc", Curso("Ninguno", 0))
    fun imprimirDatos() {
        println("Nombre: $nombre")
        println("Edad: $edad")
        println("Dirección: $dirección")
        imprimirEstadoEdad()
        println(curso.infoCurso())
    }
    fun esMenorDeEdad(): Boolean {
        return edad < 18
    }
    fun imprimirEstadoEdad() {
        val estado = if (esMenorDeEdad()) "es menor" else "mayor de edad"
        println("Estado: Es $estado.")
    }
}
fun main() {
    val cursoKotlin = Curso("Kotlin Básico", 6)
    val cursoJava = Curso("Java Intermedio", 8)
    val estudiante1 = Estudiante("Pol", 16, "C. Mayor", cursoKotlin)
    val estudiante2 = Estudiante("Eli", 20, "Enmedio 89", cursoJava)
    estudiante1.imprimirDatos()
    println()
    estudiante2.imprimirDatos()
}

```

10.4. Acceso a métodos y propiedades

En Kotlin todo es público por defecto. Para indicar **quién puede ver o usar** una clase, propiedad o método desde fuera de la clase se utilizan los **modificadores de acceso**.

```

class CuentaBancaria(val titular: String) {
    private var saldo: Double = 0.0
    fun depositar(cantidad: Double) {
        saldo += cantidad
    }
    fun mostrarSaldo() {
        println("Saldo actual: $saldo")
    }
}
fun main() {
    val cuenta = CuentaBancaria("María")
    cuenta.depositar(100.0)
    cuenta.mostrarSaldo()
    // cuenta.saldo = 1000.0 // ERROR: saldo es private
}

```

10.5. Herencia

La **Herencia** es un mecanismo por el cual una **clase (subclase)** puede **heredar propiedades y métodos de otra clase (superclase)**. Por seguridad, en Kotlin: - Las clases **no se pueden heredar** a menos que se marquen con `open`. - Los métodos también deben ser `open` para poder sobrescribirse.

```

open class Animal(val nombre: String) {
    open fun hacerSonido() {
        println("$nombre hace un sonido genérico")
    }
}
class Perro(nombre: String) : Animal(nombre) {
    override fun hacerSonido() {
        super.hacerSonido()
        println("$nombre dice: ¡Guau!")
    }
}
class Gato(nombre: String) : Animal(nombre) {
    override fun hacerSonido() {
        super.hacerSonido()
        println("$nombre dice: ¡Miau!")
    }
}
fun main() {
    val animal = Animal("Criatura")
    val perro = Perro("Firulais")
    val gato = Gato("Misu")
    animal.hacerSonido()
    println("----")
    perro.hacerSonido()
    println("----")
    gato.hacerSonido()
}

```

10.6. Arrays y ArrayLists de objetos

- Ejemplo 1 - Array de objetos:

```
fun main() {
    val animales: Array<Animal> = arrayOf(
        Perro("Rocky"),
        Gato("Luna"),
        Perro("Max"),
        Animal("Criatura misteriosa"),
        Gato("Nina")
    )
    for (animal in animales) {
        animal.hacerSonido()
    }
}
```

- Ejemplo 2 - ArrayList de objetos:

```
fun main() {
    val animales = mutableListOf<Animal>(
        Perro("Rocky"),
        Gato("Luna"),
        Perro("Max"),
        Animal("Criatura misteriosa"),
        Gato("Nina")
    )
    println("Todos los animales hacen sonido:")
    for (animal in animales) {
        animal.hacerSonido()
    }
    println("\nSolo los perros:")
    val soloPerros = animales.filterIsInstance<Perro>()
    for (perro in soloPerros) {
        perro.hacerSonido()
    }
}
```

10.7. Funciones de extensión

Las **funciones de extensión** permiten añadir nuevas funciones a clases existentes sin tener que modificarlas ni heredar de ellas.

- Ejemplo 1 - Función de extensión para la clase String:

```
fun String.saludar(): String {
    return "Hola, $this"
}
fun main() {
    val nombre = "Pol"
    println(nombre.saludar())
}
```

- Ejemplo 2 - Función de extensión para una clase personalizada:

```
open class Animal(val nombre: String) {
    open fun hacerSonido() {
        println("$nombre hace un sonido genérico")
    }
}
fun Animal.nombreEnMayusculas(): String {
    return nombre.uppercase()
}
fun main() {
    val gato = Animal("Misu")
    println(gato.nombreEnMayusculas())
}
```

- Ejemplo 3 - Función de extensión con lógica adicional:

```
fun Int.esPar(): Boolean {
    return this % 2 == 0
}
fun main() {
    val numero = 4
    if (numero.esPar()) {
        println("$numero es par")
    } else {
        println("$numero es impar")
    }
}
```

10.8. data class

Una `data class` es una clase pensada para almacenar datos sin necesidad de implementar funcionalidades.

```
data class Libro(val titulo: String, val autor: String, val any: Int)
fun main() {
    val libro1 = Libro("1984", "George Orwell", 1949)
    val libro2 = Libro("Cien años de soledad", "Gabriel García Márquez", 1967)
    val libro3 = Libro("Fahrenheit 451", "Ray Bradbury", 1953)
    val libro4 = Libro("Orgullo y prejuicio", "Jane Austen", 1813)
    val biblioteca = listOf(libro1, libro2, libro3, libro4)
    println("Biblioteca completa:")
    for (libro in biblioteca) {
        println(libro.titulo + " - " + libro.autor + " (" + libro.any + ")")
    }
    println("\nLibros ordenados por año (de menor a mayor):")
    val ordenados = biblioteca.sortedBy { it.any }
    for (libro in ordenados) {
        println(libro.titulo + " -> " + libro.any)
    }
    println("\nCopiar un libro y cambiar su año:")
    val libroModificado = libro1.copy(any = 2025)
    println("Original: " + libro1.toString())
    println("Modificado: " + libroModificado.toString())
    println("\n¿Son iguales los objetos? " + (libro1 == libroModificado))
}
```

10.9. Sobrecarga de operadores

La sobrecarga de operadores permite definir o personalizar el comportamiento de los operadores (+, -, *, ==, etc.) al aplicarse sobre instancias de nuestras propias clases.

- Ejemplo 1 - Clase Punto con operador + sobrecargado:

```
data class Punto(val x: Int, val y: Int) {
    operator fun plus(otro: Punto): Punto {
        return Punto(this.x + otro.x, this.y + otro.y)
    }
}
fun main() {
    val p1 = Punto(2, 3)
    val p2 = Punto(4, 1)
    val resultado = p1 + p2
    println(resultado)
}
```

- Ejemplo 2 - Comparar dos animales por nombre:

```
class Animal(val nombre: String) {
    override operator fun equals(other: Any?): Boolean {
        return other is Animal && this.nombre == other.nombre
    }
}
fun main() {
    val a1 = Animal("Luna")
    val a2 = Animal("Luna")
    val a3 = Animal("Max")
    println(a1 == a2)
    println(a1 == a3)
}
```

11. Funciones lambda

Las funciones lambda son funciones definidas sin un nombre, utilizadas para operaciones simples y rápidas, mejorando la legibilidad del código y el rendimiento.

- Ejemplo básico:

```
val saludar = { nombre: String -> println("Hola, $nombre") }
saludar("Pol")
```

- Ejemplo sin parámetros:

```
val decirHola = { println("Hola") }
decirHola()
```

- Ejemplo con múltiples parámetros:

```
val sumar = { a: Int, b: Int -> a + b }
println(sumar(3, 4))
```

- Ejemplo 1 - forEach con IntArray:

```
val numeros = intArrayOf(1, 2, 3, 4, 5)
numeros.forEach { println(it) }
```

- Ejemplo 2 - map para transformar:

```
val numeros = intArrayOf(1, 2, 3)
val dobles = numeros.map { it * 2 }
println(dobles)
```

- Ejemplo 3 - map + toIntArray():

```
val numeros = intArrayOf(1, 2, 3)
val doblesArray = numeros.map { it * 2 }.toIntArray()
```

- Ejemplo 4 - filter para filtrar elementos:

```
val numeros = intArrayOf(1, 2, 3, 4, 5)
val pares = numeros.filter { it % 2 == 0 }
println(pares)
```

12. Excepciones

Una excepción es un error que ocurre en tiempo de ejecución y que interrumpe el flujo normal del programa.

```
fun main() {
    try {
        val resultado = 10 / 0
        println("Resultado: $resultado")
    } catch (e1: ArithmeticException) {
        println("Error: División entre cero")
    } catch (e2: Exception) {
        println("Otro error")
    } finally {
        println("Fin del bloque try-catch")
    }
}
```

13. Package e import

Un **package** es una forma de **agrupar clases, funciones, objetos y otros archivos** relacionados bajo un mismo nombre. La palabra clave **import** se usa para **acceder a clases, funciones u objetos definidos en otros paquetes**.

- Ejemplo 1 - package e import:

```
package com.ejemplo.app
import com.ejemplo.util.saludar
fun main() {
    println(saludar("Mundo"))
}
```

- Ejemplo 2 - import ... as ... (alias):

```
import com.ejemplo.util.saludar as saludoUtil
fun main() {
    println(saludoUtil("Pol"))
}
```

- Ejemplo 3 - Importar todo un paquete (*):

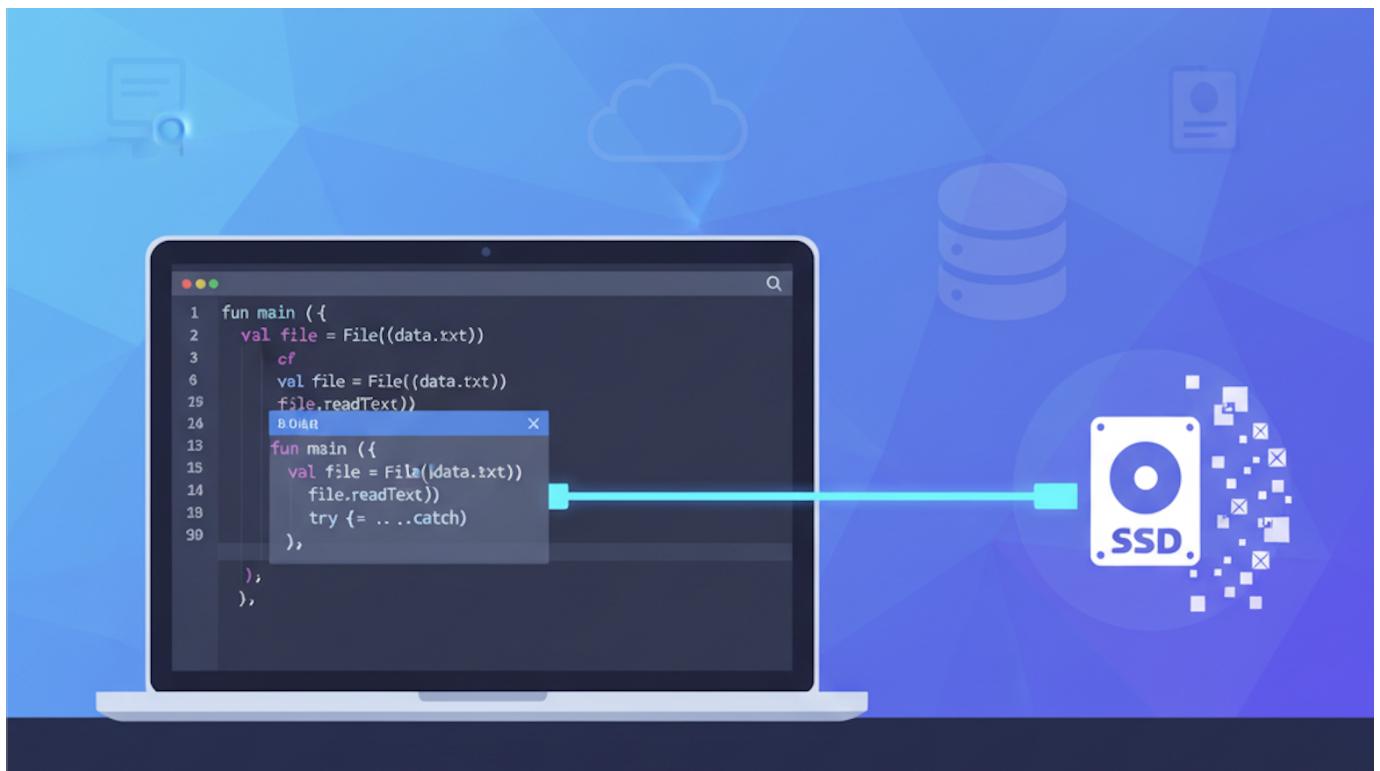
```
import com.ejemplo.util.*
```

• **Ejemplo completo:**

```
package util
fun saludar(nombre: String): String {
    return "¡Hola, $nombre!"
}
fun despedir(nombre: String): String {
    return "Adiós, $nombre."
}
package util
fun sumar(a: Int, b: Int): Int = a + b
fun multiplicar(a: Int, b: Int): Int = a * b
package app
import util.saludar
import util.despedir
import util.sumar
import util.multiplicar
fun main() {
    val nombre = "Kotlin"
    println(saludar(nombre))
    println("Suma: ${sumar(3, 5)}")
    println("Multiplicación: ${multiplicar(4, 6)}")
    println(despedir(nombre))
}
```

2.2 UD2 - Persistencia en ficheros

2.2.1 UD2 - Persistencia en ficheros



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).

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:** "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 para 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 (el profesor te dará sugerencias de mejora) o finales (el profesor 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, videos, 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).

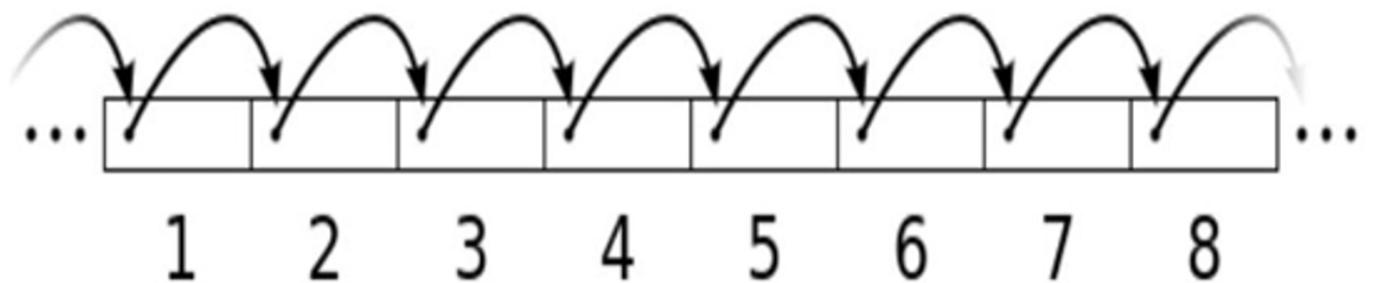
API PARA MANEJO DE FICHEROS:

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.

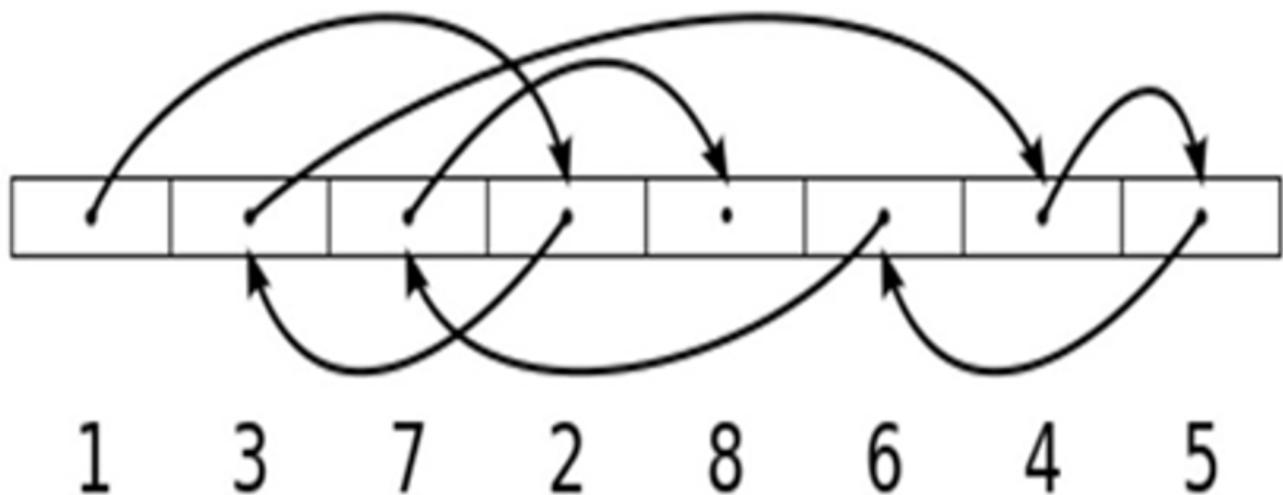
FORMAS DE ACCESO:

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.

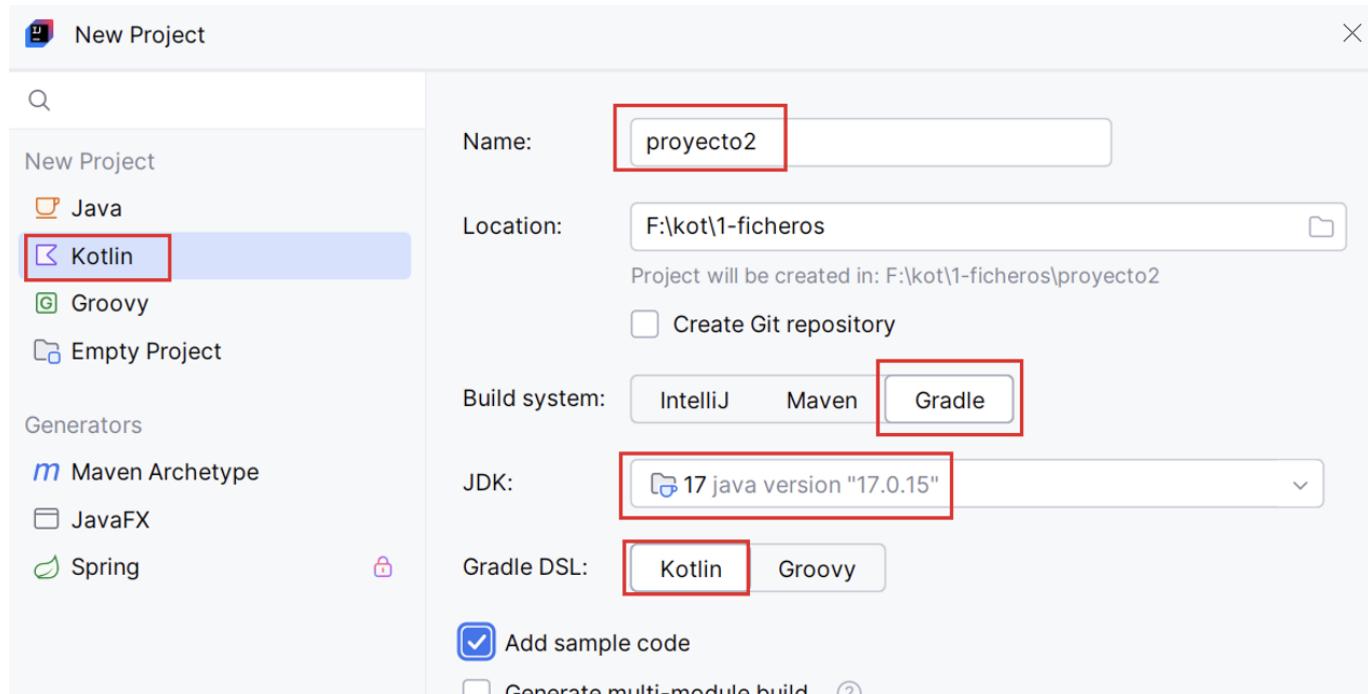


- **Acceso aleatorio:** Permite saltar a una posición concreta del fichero sin necesidad de leer lo anterior. Es útil cuando los registros tienen un tamaño fijo y se necesita eficiencia (por ejemplo, ir directamente al registro 100). Requiere técnicas más avanzadas como el uso de `FileChannel`, `SeekableByteChannel`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

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

2.2.2 Ficheros y directorios

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.

MÉTODOS PRINCIPALES DE PATH

Método	Descripción
<code>Path.of(String)</code>	Crea un objeto <code>Path</code> a partir de un <code>String</code> de ruta (Java 11+). Por debajo llama a <code>Paths.get()</code> que es el método original de la clase <code>Paths</code> (Java 7+).
<code>toString()</code>	Devuelve la ruta como un <code>String</code> (se llama por defecto desde <code>println</code>).
<code>toAbsolutePath()</code>	Devuelve la ruta absoluta del <code>Path</code> .
<code>fileName()</code>	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)
    println ("Ruta absoluta: " + rutaRelativa.toAbsolutePath())
    println ("Ruta absoluta: " + rutaAbsolutaWin)
    println ("Ruta absoluta: " + rutaAbsolutaNix)
}
```

💡 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`).

MÉTODOS PRINCIPALES DE FILES

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

💡 Crea una carpeta, dentro de tu proyecto llamada `multimedia` y guarda diferentes archivos (`pdf, jpg, txt, etc.`). Ejecuta el ejemplo anterior y comprueba que la salida es parecida a 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 peliculas.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] 2019101_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

Prepara la estructura de tu proyecto. Crea la ruta `proyecto/datos`. Basándote en los ejemplos anteriores, desarrolla un programa en tu proyecto haga lo siguiente:

- Defina dos rutas:** una para una carpeta llamada `datos_ini` y otra para una carpeta llamada `datos_fin` (ambas dentro de la carpeta `proyecto/datos` de tu proyecto).
- Comprueba los directorios:** Si las carpetas no existen las deberá crear utilizando `Files.createDirectories`.
- Añade ficheros:** Añade (manualmente y vacío) el fichero `mis_datos.json` dentro de la carpeta `datos_ini`.
- Comprueba ficheros:** Despues de la comprobación de la existencia del fichero de datos dentro de la carpeta `datos_ini` (`mis_datos.json`) imprimirá un mensaje por consola mostrando la estructura de directorios y ficheros.

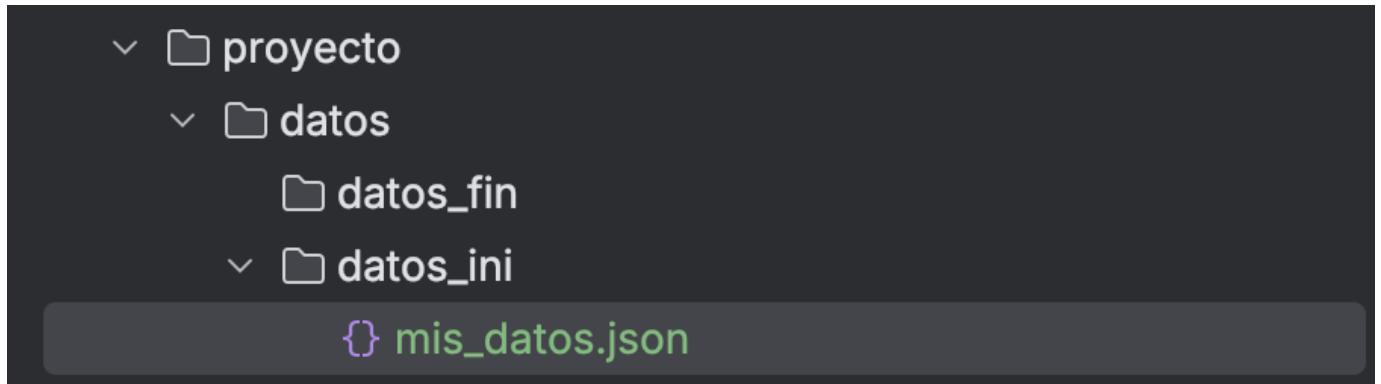
La salida de tu programa debe ser parecida a esta, la primera vez que se ejecuta:

```
CREACIÓN DE RUTAS PROYECTO
Creando rutas...
Creación de ruta para DATOS_INI
Creación de ruta para DATOS_FIN
MOSTRANDO ESTRUCTURA DE DIRECTORIOS Y FICHEROS
[DIR] datos
    [DIR] datos_fin
    [DIR] datos_ini
```

Y la segunda vez que se ejecuta, tras añadir el fichero `mis_datos.json`:

```
CREACIÓN DE RUTAS PROYECTO
  Creando rutas...
MOSTRANDO ESTRUCTURA DE DIRECTORIOS Y FICHEROS
[DIR] datos
  [DIR] datos_fin
  [DIR] datos_ini
  [FILE] mis_datos.json
```

La estructura en tu proyecto debe ser parecida a esta:



2.2.3 Ficheros de texto

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:

MÉTODOS DE FICHEROS DE TEXTO

Método	Descripción
<code>Files.readAllLines(path)</code> devuelve <code>List<String></code>	Leer ficheros.
<code>Files.exists(path)</code>	Verificar existencia.
<code>split()</code> , <code>trim()</code> , <code>toIntOrNull()</code>	Procesar texto.
<code>Files.write(path, lines)</code>	Escribe una lista de líneas (<code>List<String></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.
<code>Files.readString(Path)</code> (Java 11+), <code>Files.writeString(Path, String)</code>	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 - ESCRITURA Y LECTURA EN FICHERO 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 linea",
        "Segunda linea",
        "¡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 leido con readAllLines:")
    for (lineas in lineasLeidas) {
        println (lineas)
    }
    //readString
    val contenido = Files.readString(ruta)
    println ("Contenido leido con readString:")
    println (contenido)
    //newBufferedReader
    Files.newBufferedReader(ruta).use { reader -
        println ("Contenido leido con newBufferedReader:")
        reader.lineSequence().forEach { println (it) }
    }
}
```

```
    }  
}
```

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

```
Fichero de texto escrito.  
Contenido leido con readAllLines:  
Primera linea  
Segunda linea  
¡Hola desde Kotlin!  
Contenido leido con readString:  
Primera linea  
Segunda linea  
¡Hola desde Kotlin!  
  
Contenido leido con newBufferedReader:  
Primera linea  
Segunda linea  
¡Hola desde Kotlin!
```

2.2.4 Ficheros de intercambio

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:

MÉTODOS DE SERIALIZACIÓN/DESERIALIZACIÓN

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 - 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 = "multimedia/persona.obj"
    val rutaUsuario = "multimedia/usuario.obj"
    // Asegurar que el directorio existe
    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).

MÉTODOS DE KOTLIN-CSV

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

EJEMPLO DE 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.doyaaaaaken: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.doyaaaaaken.kotlincsv.dsl.csvReader
import com.github.doyaaaaaken.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) //crear el objeto Planta
                } 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 {
                // Si la fila no tiene suficientes columnas, es inválida. Devolvemos 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

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.

MÉTODOS DE JACKSON XML

Método	Descripción
<code>readValue(File, Class<T>)</code>	Lee un fichero XML y lo convierte en un objeto Kotlin/Java.
<code>readValue(String, Class<T>)</code>	Lee un String XML y lo convierte en un objeto.
<code>writeValue(File, Object)</code>	Escribe un objeto como XML en un fichero.
<code>writeValueAsString(Object)</code>	Convierte un objeto en una cadena XML.
<code>writeValueAsBytes(Object)</code>	Convierte un objeto en un array de bytes XML.
<code>registerModule(Module)</code>	Registra un módulo como <code>KotlinModule</code> o <code>JavaTimeModule</code> .
<code>enable(SerializationFeature)</code>	Activa una opción de serialización (por ejemplo, indentado).
<code>disable(DeserializationFeature)</code>	Desactiva una opción de deserialización.
<code>configure(MapperFeature, boolean)</code>	Configura opciones generales del mapeo.
<code>setDefaultPrettyPrinter(...)</code>	Establece un formateador personalizado.

EJEMPLO DE 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_rie go>7</frecuencia_rie go>
    <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_rie go>3</frecuencia_rie go>
    <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_rie go>5</frecuencia_rie go>
    <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 annus</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> (asi 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 raiz
@JacksonXmlRootElement(localName = "plantas")
// Data class que representa el elemento raiz 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> {
    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

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).

MÉTODOS DE KOTLINX.SERIALIZATION

Método / Ejemplo	Descripción
Json.encodeToString(objeto)	Json.encodeToString(persona)
Json.encodeToString(serializer, obj)	Json.encodeToString(Persona.serializer(), persona)
Json.decodeFromString(json)	Json.decodeFromString<Persona>(json)
Json.decodeFromString(serializer, s)	Json.decodeFromString(Persona.serializer(), json)
Json.encodeToJsonElement(objeto)	val elem = Json.encodeToJsonElement(persona)
Json.decodeFromJsonElement(elem)	val persona = Json.decodeFromJsonElement<Persona>(elem)
Json.parseToJsonElement(string)	val elem = Json

EJEMPLO DE 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_riege": 7,
    "altura_maxima": 0.6
  },
  {
    "id_planta": 2,
    "nombre_comun": "Lavanda",
    "nombre_cientifico": "Lavandula angustifolia",
    "frecuencia_riege": 3,
    "altura_maxima": 1.0
  },
  {
    "id_planta": 3,
    "nombre_comun": "Helecho de Boston",
    "nombre_cientifico": "Nephrolepis exaltata",
    "frecuencia_riege": 5,
    "altura_maxima": 0.9
  },
  {
    "id_planta": 4,
    "nombre_comun": "Bambú de la suerte",
    "nombre_cientifico": "Dracaena sanderiana",
    "frecuencia_riege": 4,
    "altura_maxima": 1.5
  },
  {
    "id_planta": 5,
    "nombre_comun": "Girasol",
    "nombre_cientifico": "Helianthus annuus",
    "frecuencia_riege": 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_riege: 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_riege} 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
```

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

💡 Realiza algunas conversiones entre ficheros CSV, JSON y XML para practicar la lectura / escritura y la serialización / deserialización. Puedes reutilizar el código de los ejemplos.

⌚ PRÁCTICA 3: CREACIÓN Y LECTURA DE UN FICHERO DE DATOS

Realiza lo siguiente:

- **Diseña tu data class:** Define la `data class` de Kotlin que represente un único elemento de tu colección de datos. Debe tener un ID único de tipo `Int`, un nombre de tipo `String` y, al menos, otros dos campos (al menos uno de tipo `Double`).
- **Crea tu fichero de datos:** (`.csv`, `.json` o `.xml`) con al menos 5 registros de tu colección dentro de la carpeta `datos_ini`.
- **Añade dependencias necesarias:** Añade las librerías necesarias para leer tu fichero y *serializar / deserializar* datos en `build.gradle.kts`.
- **Crea la función de lectura:** La función debe leer el fichero de texto y devolver una lista de objetos `leerDatosIniciales(): List<DataClass>`.
- **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 el profesor te dé sugerencias de mejora (el programa entregado deberá ejecutarse, si da error de ejecución, no se podrá revisar).

2.2.5 Ficheros binarios

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.

MÉTODOS DE FICHEROS BINARIOS NO ESTRUCTURADOS

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

EJEMPLO BINARIO NO ESTRUCTURADO:

Escribir bit a bit los datos 1 2 3 4 5 en un fichero 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 'documentos' 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: $ruta")
            } else {
                // Lectura del fichero binario
                val bytes = Files.readAllBytes(ruta)
                println ("Contenido leido (byte a byte):")
                for (b in bytes) {
                    print ("$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 leido (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

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

MÉTODOS DE DATAINPUTSTREAM

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

EJEMPLO:

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 leido:")
    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 leido:
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.

MÉTODOS DE FICHEROS DE IMÁGENES

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

EJEMPLO QUE GENERA 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.absolutePath}")
}

```

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

EJEMPLO QUE CONVIERTE UNA IMAGEN A ESCALA DE GRISSE:

```

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, pixel por pixel
for (x in 0 until imagen.width) {
    for (y in 0 until imagen.height) {
        // Obtenemos el color del pixel 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 pixel 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.

2.2.6 Ficheros de acceso aleatorio

6. Ficheros de acceso aleatorio

Un fichero 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

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

MÉTODOS DE BYTEBUFFER

Método	Descripción
<code>allocate(capacidad)</code>	Crea un buffer con capacidad fija en memoria (no compartida).
<code>wrap(byteArray)</code>	Crea un buffer que envuelve un array de bytes existente (memoria compartida).
<code>wrap(byteArray, offset, length)</code>	Crea un buffer desde una porción del array existente.
<code>put(byte), putInt(int), putDouble(double), putFloat(float), putChar(char), putShort(short), putLong(long)</code>	Escribe un byte, int, double, float, char, short o long en la posición actual.
<code>put(byte[], offset, length)</code>	Escribe una porción de un array de bytes.
<code>get(), getInt(), getDouble(), getFloat(), getChar(), getShort(), getLong()</code>	Lee un byte, int, double, float, char, short o long desde la posición actual.
<code>get(byte[], offset, length)</code>	Lee una porción del buffer a un array.

MÉTODOS DE CONTROL DEL BUFFER

Método	Descripción
<code>position()</code>	Devuelve la posición actual del cursor.
<code>position(int)</code>	Establece la posición del cursor.
<code>limit()</code>	Devuelve el límite del buffer.
<code>limit(int)</code>	Establece un nuevo límite.
<code>capacity()</code>	Devuelve la capacidad total del buffer.
<code>clear()</code>	Limpia el buffer: posición a 0, límite al máximo (sin borrar contenido).
<code>flip()</code>	Prepara el buffer para lectura después de escribir.
<code>rewind()</code>	Posición a 0 para releer desde el inicio.
<code>remaining</code>	Indica cuántos elementos quedan por procesar.
<code>hasRemaining()</code>	<code>true</code> si aún queda contenido por leer o escribir.

IMPORTANTE: un fichero .dat no es un fichero de texto. No se puede abrir con el Bloc de Notas,TextEdit, o un editor de código en modo texto normal. Si se abre con estos programas se ve una mezcla de caracteres extraños, símbolos y espacios ("basura"). Hay herramientas online y plugins para los IDE para poder abrir los ficheros y ver la información en binario que contienen.

EJEMPLO:

El siguiente ejemplo utiliza `FileChannel` y `ByteBuffer` para crear y leer un fichero llamado `mediciones.dat` con registros con la siguiente estructura:

- ID del sensor (Int - 4 bytes)
- temperatura (Double - 8 bytes)
- humedad (Double - 8 bytes)

A continuación se muestra el código con las funciones para añadir una medición al final del fichero y leer todas las mediciones que hay en él.

```

import java.nio.ByteBuffer // "contenedor" de bytes en memoria.
import java.nio.ByteOrder // especificar el orden de los bytes
import java.nio.channels.FileChannel // canal que conecta con el fichero
import java.nio.file.Files
import java.nio.file.Path
import java.nio.file.StandardOpenOption
const val TAMANO_ID = Int.SIZE_BYTES // 4 bytes
const val TAMANO_NOMBRE = 20 // String de tamaño fijo 20 bytes
const val TAMANO_TEMPERATURA = Double.SIZE_BYTES // 8 bytes
const val TAMANO_HUMEDAD = Double.SIZE_BYTES // 8 bytes
const val TAMANO_REGISTRO = TAMANO_ID + TAMANO_NOMBRE + TAMANO_TEMPERATURA + TAMANO_HUMEDAD
fun main() {
    val rutaFichero = Path.of("mediciones.dat")
    escribirMedicion(rutaFichero, 101, "Atenea", 25.5, 60.2)
    escribirMedicion(rutaFichero, 102, "Hera", 26.1, 58.9)
    escribirMedicion(rutaFichero, 103, "Iris", 28.4, 65.9)
    escribirMedicion(rutaFichero, 104, "Selene", 28.4, 65.9)
    leerMediciones(rutaFichero) // leer todas las mediciones
}
// Función que escribe una medición en el fichero.
fun escribirMedicion(ruta: Path, idSensor: Int, nombre: String, temperatura: Double, humedad: Double) {
    /* .use { ... } abre el canal (se cerrará automáticamente al final del bloque) Escribir con APPEND para añadir el final */
    FileChannel.open(ruta, StandardOpenOption.WRITE, StandardOpenOption.CREATE, StandardOpenOption.APPEND).use { canal ->
        // Crear un ByteBuffer de nuestro tamaño y especificamos el orden de bytes
        val buffer = ByteBuffer.allocate(TAMANO_REGISTRO)
        buffer.order(ByteOrder.nativeOrder())
        /* Escribimos los datos en el buffer en el orden correcto.
        'put' avanza la "posición" interna del buffer.*/
        buffer.putInt(idSensor) // Escribe 4 bytes
        /* Para escribir el String hay que convertirlo a un array de bytes de tamaño fijo.
        Inicializamos el array de bytes rellenándolo con el carácter espacio.
        '.code.toByte()' convierte el carácter espacio a su valor de byte.*/
        val nombreCompleto = ByteArray(TAMANO_NOMBRE) { ' '.code.toByte() }
        // Convertimos el String de entrada a un array de bytes temporal.
        val nombreBytes = nombre.toByteArray(Charsets.UTF_8)
        /* Copiamos los bytes del String al principio de nuestro array de tamaño fijo.
        Si 'nombre' ocupa menos de 20 bytes, el resto de 'nombreCompleto' seguirá lleno de espacios.
        Si 'nombre' ocupa más de 20 bytes, solo se copiarán los primeros 20.*/
        nombreBytes.copyOfInto(nombreCompleto)
        buffer.put(nombreCompleto) // Escribe 20 bytes
        buffer.putDouble(temperatura) // Escribe 8 bytes
        buffer.putDouble(humedad) // Escribe 8 bytes
        /* 'flip()' prepara el buffer para ser leído o escrito
        Resetea la 'posición' a 0 y limita al tamaño total
        El canal escribirá desde la posición 0 hasta la 20 */
        buffer.flip()
        // Escribimos el contenido del buffer en el fichero a través del canal.
        canal.write(buffer)
        println ("Medición (ID: $idSensor) escrita correctamente.")
    }
}
// Función que lee TODAS las mediciones almacenadas en el fichero.
fun leerMediciones(ruta: Path) {
    if (!Files.exists(ruta)) {
        println ("El fichero ${ruta.fileName} no existe. No hay nada que leer.")
    } else {
        println ("\n--- Leyendo todas las mediciones ---")
        FileChannel.open(ruta, StandardOpenOption.READ).use { canal ->
            // Crear buffer
            val buffer = ByteBuffer.allocate(TAMANO_REGISTRO)
            buffer.order(ByteOrder.nativeOrder())
            /* Leer del canal en un bucle hasta que se alcance el final del fichero.
            canal.read(buffer) lee bytes del fichero y los guarda en el buffer.
            Devuelve el número de bytes leídos, o -1 si ya no hay más datos.*/
            while (canal.read(buffer) > 0) {
                /* Después de 'canal.read()' su posición está al final.
                'flip()' resetear la posición a 0. Para poder leer
                los datos que acabamos de cargar desde el principio del buffer. */
                buffer.flip()
                // Leemos los datos en el mismo orden en que los escribimos.
                val id = buffer.getInt()
                // Crear un array de bytes vacío para guardar los datos del nombre.
                val nombreCompleto = ByteArray(TAMANO_NOMBRE)
                // Leer 20 bytes del buffer y los guardamos en nuestro array.
                buffer.get(nombreCompleto)
                /* Convertir el array de bytes a un String. Usar .trim() para eliminar
                los espacios en blanco que se escribieron al final */
            }
        }
    }
}

```

```
        val nombre = String(nombreCompleto, Charsets.UTF_8).trim()
        val temp = buffer.getDouble()
        val hum = buffer.getDouble()
        println (" - ID: $id, Nombre: $nombre, Temperatura: $temp °C, Humedad: $hum %")
        // clear() resetea la posición a 0 y el límite a la capacidad total.
        buffer.clear()
    }
}
}
```



```
Medición (ID: 101) escrita correctamente.  
Medición (ID: 102) escrita correctamente.  
Medición (ID: 103) escrita correctamente.  
Medición (ID: 104) escrita correctamente.  
  
--- Leyendo todas las mediciones ---  
- ID: 101, Nombre: Atenea, Temperatura: 25.5 °C, Humedad: 60.2 %  
- ID: 102, Nombre: Hera, Temperatura: 26.1 °C, Humedad: 58.9 %  
- ID: 103, Nombre: Iris, Temperatura: 28.4 °C, Humedad: 65.9 %  
- ID: 104, Nombre: Selene, Temperatura: 28.4 °C, Humedad: 65.9 %
```

Ahora que ya tenemos la información guardada en nuestro fichero `.dat` y sabemos leerla, vamos a ampliar la aplicación con una función que recoge el ID del sensor a modificar y los nuevos datos de temperatura y humedad. Cuando localiza el registro del sensor cuyo ID coincide con el buscado, escribe los nuevos datos en las posiciones de los bytes correspondientes.

```

    fun actualizarMedicion(ruta: Path, idSensorBuscado: Int, nuevaTemperatura: Double, nuevaHumedad: Double) {
        if (!Files.exists(ruta)) {
            println ("Error: El fichero no existe, no se puede actualizar.")
        } else {
            println ("\nIntentando actualizar medición para ID: $idSensorBuscado...")
            FileChannel.open(ruta, StandardOpenOption.READ, StandardOpenOption.WRITE).use { canal ->
                // Creamos un buffer pequeño, solo para leer el ID en cada iteración.
                // No necesitamos cargar el registro completo solo para buscar.
                val buffer = ByteBuffer.allocate(TAMANO_ID)
                buffer.order(ByteOrder.nativeOrder())
                var posicionActual: Long = 0
                var encontrado = false
                while (canal.position() < canal.size() && !encontrado) {
                    // Guardar la posición del inicio del registro que estamos a punto de leer.
                    posicionActual = canal.position()

                    // Limpiamos y leemos solo los 4 bytes del ID.
                    buffer.clear()
                    canal.read(buffer)

                    // Preparamos el buffer para leer el entero.
                    buffer.flip()
                    val idActual = buffer.getInt()
                    println ("leyendo ID: " + idActual)
                    // Comparamos el ID leído con el que estamos buscando.
                    if (idActual == idSensorBuscado) {
                        encontrado = true
                        println ("Sensor $idSensorBuscado en posición $posicionActual.")
                        // Posición temperatura = inicio registro + tamaño del ID + tamaño nombre
                        canal.position(posicionActual + TAMANO_ID + TAMANO_NOMBRE)
                        val bufferDatos = ByteBuffer.allocate(TAMANO_TEMPERATURA + TAMANO_HUMEDAD)
                        bufferDatos.order(ByteOrder.nativeOrder())
                        bufferDatos.putDouble(nuevaTemperatura)
                        bufferDatos.putDouble(nuevaHumedad)
                        bufferDatos.flip()
                        canal.write(bufferDatos)
                        println ("Medición actualizada con éxito a Temp: $nuevaTemperatura, Hum: $nuevaHumedad.")
                    } else {
                        canal.position(posicionActual + TAMANO_REGISTRO)
                    }
                }
                if (!encontrado) {
                    println ("Medición con ID: $idSensorBuscado no encontrada")
                }
            }
        }
    }
}

```

La llamada a esta nueva función en el main podría ser:

```
actualizarMedicion(rutaEichero 102 21 0 72 3)
```

Se vuelve a llamar a `leerMediciones` para comprobar que la información del sensor se ha modificado correctamente:

```
leerMediciones(rutaFichero)
```

Realiza los siguientes pasos:

- Añade el código de la función `actualizarMedicion()` al proyecto del ejemplo anterior.
- Comenta en el `main` las llamadas a la función `escribirMedicion()`.
- Añade al `main` las llamadas a `actualizarMedicion()` y a `leerMediciones()`.
- Ejecuta la aplicación y comprueba que la salida es la siguiente:

```
--- Leyendo todas las mediciones ---
- ID: 101, Nombre: Atenea, Temperatura: 25.5 °C, Humedad: 60.2 %
- ID: 102, Nombre: Hera, Temperatura: 26.1 °C, Humedad: 58.9 %
- ID: 103, Nombre: Iris, Temperatura: 28.4 °C, Humedad: 65.9 %
- ID: 104, Nombre: Selene, Temperatura: 28.4 °C, Humedad: 65.9 %

Intentando actualizar medición para ID: 102...
leyendo ID: 101
leyendo ID: 102
Sensor 102 en posición 40.
Medición actualizada con éxito a Temp: 21.0, Hum: 72.3.

--- Leyendo todas las mediciones ---
- ID: 101, Nombre: Atenea, Temperatura: 25.5 °C, Humedad: 60.2 %
- ID: 102, Nombre: Hera, Temperatura: 21.0 °C, Humedad: 72.3 %
- ID: 103, Nombre: Iris, Temperatura: 28.4 °C, Humedad: 65.9 %
- ID: 104, Nombre: Selene, Temperatura: 28.4 °C, Humedad: 65.9 %
```

Por último, ampliaremos la aplicación para poder eliminar los datos de un sensor 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 eliminarMedicion(ruta: Path, idSensorAEliminar: Int) {
    val rutaTemp = Path.of("temp.dat")
    if (!Files.exists(ruta)) {
        println ("Error: El fichero no existe, no se puede actualizar.")
    } else {
        println ("\nIntentando eliminar medición para el sensor con ID: $idSensorAEliminar...")
        FileChannel.open(ruta, StandardOpenOption.READ).use { canal ->
            // Crear buffer
            val buffer = ByteBuffer.allocate(TAMANO_REGISTRO)
            buffer.order(ByteOrder.nativeOrder())
            /* Leer del canal en un bucle hasta que se alcance el final del fichero.
            canal.read(buffer) lee bytes del fichero y los guarda en el buffer.
            Devuelve el número de bytes leídos, o -1 si ya no hay más datos.*/
            while (canal.read(buffer) > 0) {
                /* Después de `canal.read()` su posición está al final.
                `flip()` resetear la posición a 0. Para poder leer
                los datos que acabamos de cargar desde el principio del buffer. */
                buffer.flip()
                // Leemos los datos en el mismo orden en que los escribimos.
                val id = buffer.getInt()
                // Crear un array de bytes vacío para guardar los datos del nombre.
                val nombreCompleto = ByteArray(TAMANO_NOMBRE)
                // Leer 20 bytes del buffer y los guardamos en nuestro array.
                buffer.get(nombreCompleto)
                /* Convertir el array de bytes a un String. Usar .trim() para eliminar
                los espacios en blanco que se escribieron al final */
                val nombre = String(nombreCompleto, Charsets.UTF_8).trim()
                val temp = buffer.getDouble()
                val hum = buffer.getDouble()
                if (id!=idSensorAEliminar) {
                    // Usar nuestra función para escribir en el fichero temporal
                    escribirMedicion(rutaTemp, id, nombre, temp, hum)
                }
                buffer.clear()
            }
        }
        Files.delete(ruta) //borrar fichero original
        Files.move(rutaTemp, ruta) // renombrar temporal
    }
}
```

La llamada a esta nueva función en el main podría ser:

```
eliminarMedicion(rutaFichero, 102)
```

Se vuelve a llamar a `leerMediciones` para comprobar que la información del sensor se ha modificado correctamente:

```
leerMediciones(rutaFichero)
```

 **Realiza los siguientes pasos:**

- Añade el código de la función `eliminarMedicion()` al ejemplo anterior.
- Comenta en el `main` la llamada a la función `actualizarMedicion()`.
- Añade al `main` la llamada a `eliminarMedicion()`.

Ejecuta la aplicación y comprueba que la salida es la siguiente:

```
--- Leyendo todas las mediciones ---
- ID: 101, Nombre: Atenea, Temperatura: 25.5 °C, Humedad: 60.2 %
- ID: 102, Nombre: Hera, Temperatura: 21.0 °C, Humedad: 72.3 %
- ID: 103, Nombre: Iris, Temperatura: 28.4 °C, Humedad: 65.9 %
- ID: 104, Nombre: Selene, Temperatura: 28.4 °C, Humedad: 65.9 %
```

```
Intentando eliminar medición para el sensor con ID: 102...
Medición (ID: 101) escrita correctamente.
Medición (ID: 103) escrita correctamente.
Medición (ID: 104) escrita correctamente.
```

```
--- Leyendo todas las mediciones ---
- ID: 101, Nombre: Atenea, Temperatura: 25.5 °C, Humedad: 60.2 %
- ID: 103, Nombre: Iris, Temperatura: 28.4 °C, Humedad: 65.9 %
- ID: 104, Nombre: Selene, Temperatura: 28.4 °C, Humedad: 65.9 %
```

 **PRÁCTICA 5: MODIFICAR Y ELIMINAR REGISTROS EN FICHEROS .DAT**

Realiza lo siguiente: * **Crea la función `modificarReg()`:** Pedirá al usuario el ID del registro a modificar y buscará ese registro en el fichero. Si lo encuentra, pedirá los nuevos datos. Utilizará acceso aleatorio (`FileChannel.position()`) para saltar a la posición exacta de ese registro y sobrescribir únicamente los campos modificados, sin alterar el resto del fichero. * **Crea la función `eliminarReg()`:** Debe recibir un ID y eliminar el registro correspondiente. Implementa la técnica de streaming (leer el fichero original registro a registro, escribir los que se conservan en un fichero temporal, borrar el original y renombrar el temporal). * **Comprueba:** Prueba estas funciones desde `main`, llamando a `mostrarTodo()` antes y después de cada operación para verificar los resultados.

2.2.7 Documentación final

7. Documentación: El Fichero LEEME.md

En un proyecto de software el código fuente por sí solo no cuenta toda la historia y es fundamental crear documentación adicional. La forma estándar y más extendida de hacerlo es a través de un fichero `LEEME.md` (o `README.md`). Un proyecto sin un `LEEME.md` se considera incompleto o poco profesional.

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 guardarlo 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:

```
Gestor de mediciones
Este es un programa de consola desarrollado en Kotlin para gestionar una colección de registros de temperaturas y humedad registradas por unos sensores.
Los datos se almacenan en un fichero binario de acceso aleatorio llamado *mediciones.dat*

1. Estructura de datos
Data Class:
```kotlin
data class Sensor(
    val id_sensor: Int,
    val nombre: String,
    val temperatura: Double
    val humedad: Double
)
```

```

#### ESTRUCTURA DEL REGISTRO BINARIO:

- **ID:** Int - 4 bytes
- **Nombre:** String - 20 bytes (longitud fija)
- **temperatura:** Double - 8 bytes
- **humedad:** Double - 8 bytes
- **Tamaño Total del Registro:**  $4 + 20 + 8 + 8 = 40$  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 *datos\_iniciales.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 sensores sin desperdiciar demasiado espacio.