

Tema 1 parte 2 - Lenguaje Kotlin II

Descargar estos apuntes [pdf](#) o [html](#)

Índice

▼ Colecciones

▼ Arrays

- Array Bidimensional
- Listas Mutables
- Mapas Mutables

▼ Programación Funcional

▼ Lambdas

- Definiendo y usando funciones de orden superior HOF
- Clausuras

▼ Map-Filter-Fold

- Creando objetos anónimos de consultas sobre objetos complejos
- Transformaciones de datos usando map
- Haciendo agrupaciones con groupBy
- Obteniendo secuencias anidadas con flatMap
- Ejemplo de consulta de datos mapeados a objetos en Kotlin


Colecciones

- [Colecciones en Kotlin](#)

Ya las hemos usado en algunos de los diferentes ejemplos que hemos estado viendo en el tema. Sin embargo, en Kotlin hay diferentes formas de generar **secuencias de objetos**: **arrays**, **listas inmutables**, **listas mutables**, etc.

 **Resumen:** Tabla resumen de las más comunes y cómo inicializarlas por extensión

Tipo	Descripción	Literales	Mutabilidad	Modificar tamaño
arrayOf	Array de objetos tradicional	Llista inmutable	sí	no
listOf	Llista inmutable	listOf(1, 2)	no	no
arrayListOf	Llista mutable	arrayListOf(1, 2)	sí	sí
mapOf	HashMap inmutable	mapOf(1 to "A", 2 to "B")	no	no
mutableMapOf	HashMap mutable	...	sí	sí

 **Nota:** Debemos llevar cuidado, hay que distinguir la inmutabilidad de la colección de la inmutabilidad de la variable que la contiene.

Arrays

Los **arrays** son mutable, es decir pueden cambiar el valor de sus elementos durante la ejecución, pero como ya conocemos por otros lenguajes, su tamaño no se puede cambiar una vez esté definido, existiendo otro tipo de colecciones para este uso.

👉 **Importante:** En Kotlin las declaraciones y acceso a las posiciones del array no suelen realizarse mediante corchetes, como en otros lenguajes, sino que se utilizan los métodos de la clase `Array`.

En el siguiente ejemplo podemos ver diferentes formas de recorrer el array...

```
fun main() {
    val weekDays = arrayOf("primavera", "verano", "otoño", "invierno")

    // Similar al foreach de C#
    for (dato in weekDays) println(dato)

    // Recorrido con iterador
    val it = weekDays.iterator()
    while (it.hasNext()) println(it.next())

    // Foreach con HOF
    weekDays.forEach { println(it) }

    // Acceso mediante índice
    for (i in weekDays.indices) println(weekDays[i])

    // Acceso mediante índice con until
    for (i in 0 until weekDays.size) println(weekDays[i])

    // Volcando ambos datos en una tupla
    for ((posicion, valor) in weekDays.withIndex())
        println("La posición $posicion contiene el valor $valor")
}
```

Además Kotlin distingue los arrays de primitivas usando clases propias:

`BooleanArray`, `ByteArray`, `CharArray`, `ShortArray`, `IntArray`, `FloatArray`, `DoubleArray`, y la forma de construir literales es mediante:

`booleanArrayOf`, `byteArrayOf`, `charArrayOf`, `shortArrayOf`, `intArrayOf`, `floatArrayOf`, `doubleArrayOf`

En el siguiente ejemplo se ha creado un array de float de tamaño 4, el acceso está realizado mediante `[]` además la última línea produce excepción al intentar realizar un acceso no permitido.

```
fun main() {  
    val arrayFloat = FloatArray(size = 4)  
    arrayFloat[3] = 2.5f;  
    arrayFloat.set(0, 3f);  
    print(arrayFloat[0])  
    arrayFloat[4] = 8.9f // Excepción por acceso fuera límites  
}
```

Si queremos crear un array de un tipo de objeto determinado, vamos a suponer un tipo **Persona** con el siguiente código:

```
data class Persona(val nombre: String, val edad: Int)
```

El array de personas lo podremos inicializar en el momento de creación del array, de la siguiente manera (en este caso solo tendrá un elemento):

```
fun main() {  
    val personas = arrayOf(Persona( nombre = "Ana", edad = 12))  
    println(personas[0])  
}
```

Otra opción es permitir que el array pueda tener valores nulos, para añadir posteriormente los elementos. En este caso tendremos que utilizar el operador de llamadas seguras `?.`, que solo llamará al método en el caso que el valor no sea nulo, evitando **NullPointerException**.

```
fun main() {  
    val personas = arrayOfNulls<Persona>(size = 2)  
    personas[0] = Persona(nombre = "Ana", edad = 12)  
    println(personas[0] ?. "No hay datos")  
}
```

Array Bidimensional

En kotlin los arrays de más de una dimensión se tratan como arrays de arrays (similar a las tablas dentadas de C#). Por lo que se pueden crear filas de distintos tamaños. El siguiente código crea una matriz de enteros de 4 x 4 inicializada a valor 0 y en el elemento segunda fila y segunda columna a valor 3.

```
fun main() {  
    val matriz = Array(4) { IntArray(4) }  
    matriz[1][1] = 3  
    for (e in matriz) {  
        for (i in e) {  
            print(String.format("%3d", i))  
        }  
        println()  
    }  
}
```

Si no queremos inicializar cada una de las filas cuando creamos el array, se puede hacer posteriormente si se crea la matriz anulable. En este ejemplo creamos una matriz de tres filas, cada una de ellas con tamaño de columnas distinto (2, 3 y 4 respectivamente) e inicializamos toda la matriz a valor 3.

```
fun main() {  
    val dentada = arrayOfNulls<IntArray>(3)  
    dentada[0] = IntArray(2)  
    dentada[1] = IntArray(3)  
    dentada[2] = IntArray(4)  
  
    for (i in dentada.indices)  
        for (j in 0 until dentada[i]!!.size)  
            dentada[i]!![j] = 3  
}
```

Listas Mutables

- [Listas mutables](#)

Las [listas mutables](#) están representadas por la clase `ArrayList` que implementa la interface `MutableList` y en el fondo es equivalente al tipo `ArryList` de Java.

Dispondremos de métodos similares a otros lenguajes `remove`, `removeAt`, `indexOf`, `clear`. Así como la posibilidad de obtener un **iterador** para iterar entre los elementos de la secuencia.

Formas de rellenar un lista mutable ...

```
fun main() {  
    // Definiendola por extensión  
    val personas1 = mutableListOf(  
        Persona(nombre = "Ana", edad = 12),  
        Persona(nombre = "Pedro", edad = 15)  
    )  
    // Equivalente a Java  
    val personas2 = ArrayList<Persona>()  
    personas2.add(Persona("Ana", 12))  
    personas2.add(Persona("Pedro", 15))  
    // Usando apply me permite aplicar las operaciones entre llaves al  
    // objeto que acabo de crear. Es una forma de no repetir personas.  
    val personas3 = ArrayList<Persona>().apply {  
        add(Persona("Ana", 12))  
        add(Persona("Pedro", 15))  
    }  
}
```

Formas de recorrer un lista mutable ...

```
// Foreach tradicional  
for (p in personas1) {  
    println(p)  
}  
// Foreach funcional  
personas2.forEach { p -> println(p) }  
// Usando un indizador  
for (i in personas3.indices) {  
    println(personas3[i])  
}  
// Obtengo un iterador con sus operaciones típicas.  
val it = personas3.iterator()  
while (it.hasNext()) {  
    println(it.next())  
}
```


Mapas Mutables

También tendremos mapas `MutableMap` y mapas inmutables `Map` en kotlin. El funcionamiento es similar, por lo que vamos a ver los ejemplos de los mapas mutables.

Tendremos diferentes formas de añadir pares clave-valor como sucede en otros lenguajes pero, la clase **Pair**, de la que hablamos en las funciones y que representa una **tupla con dos valores**, nos permitirá crear pares de valores que ayudarán a añadir elementos a una mapa o recorrerlo.

Además, podemos usar el operador `+=` para agregar elementos.

Formas de rellenar un mapa mutable ...

```
fun main() {  
    // Definiéndolo por extensión con el operador to  
    val personas1 = mutableMapOf(  
        "21456874L" to Persona("Ana", 12),  
        "13232345K" to Persona("Luis", 13)  
    )  
  
    // Tradicional con índices  
    val personas2 = mutableMapOf<String, Persona>()  
    personas2["21456874L"] = Persona("Ana", 12)  
    personas2["13232345K"] = Persona("Luis", 13)  
  
    // Con put como en Java usando tuplas y apply  
    val personas3 = mutableMapOf<String, Persona>().apply {  
        put("21456874L", Persona(nombre = "Ana", edad = 12))  
        put("13232345K", Persona(nombre = "Luis", edad = 13))  
    }  
  
    // Con el operador +=  
    val personas4 = mutableMapOf<String, Persona>()  
    personas3 += Pair("21456874L", Persona("Ana", 12))  
    personas3 += Pair("13232345K", Persona("Luis", 13))  
}
```

Formas de recorrer un mapa mutable ...

```
fun main() {  
    // Usando las claves y un indizador.  
    for (dni in personas1.keys) {  
        println("$dni -> ${personas1[dni]}")  
    }  
  
    // Usando tuplas clave - valor ...  
    // Con un foreach tradicional  
    for ((dni, persona) in personas3) {  
        println("$dni -> $persona")  
    }  
    // Con un iterador con sus operaciones típicas.  
    val iterador = personas3.iterator()  
    while (iterador.hasNext()) {  
        val (dni, persona) = iterador.next()  
        println("$dni -> $persona")  
    }  
    // Con un foreach funcional  
    personas3.forEach { (dni, persona) ->  
        println("$dni -> $persona")  
    }  
}
```

Programación Funcional

Vamos a ver algunas de las peculiaridades de la programación funcional en Kotlin a través de conceptos vistos en primer curso.

Lambdas

- [Lambdas en Kotlin](#)

Son tipados como sucede en C# y se definen siempre entre llaves `{ definición }` análogamente a otros lenguajes con el operador `->` por ejemplo `{ v:Int -> v % 2 == 0 }`.

✦ **Nota:** Debemos poner el tipo del parámetro si no va implícito.

Si es un procedimiento que no tiene ningún parámetro de entrada no hace falta poner `{ () -> evaluación }` sino directamente `{ evaluación }`. Por eso utilizamos las llaves.

👉 **Importante:** En ocasiones esta sintaxis entre llaves puede **llevarnos a confusión** porque `{ evaluación }` puede confundirse con un **bloque de código** cuando realmente estamos definiendo una función lambda con **los argumentos implícitos** o **sin argumentos**.

El tipo de estas funciones puede definirse de **tres formas**:

1. **SAM** ('*Single Abstract Method*') interface o también conocido como '**functional interface**': Podemos ponerle nombre al tipo y tal y como sucede en Java, es una interfaz pero con una única función abstracta. Por eso ponemos `fun` delante de la palabra reservada interface.

```
fun interface Predicat<T> {  
    fun compleix(dato: T): Boolean  
}  
  
fun main() {  
    val esPar = Predicat<Int> { v -> v % 2 == 0 }  
    println("Es par 4 = ${esPar.compleix(4)}")  
}
```

✦ **Nota:** Aunque es muy similar a Java en Kotlin es poco usada por ser muy verbosa y ser poco práctica al obligarnos a definir un tipo de forma explícita.

2. El tipo se deduce de la inicialización


```
fun main() {  
    // Fíjate que tenemos que indicar el tipo del parámetro  
    // o parámetros de entrada.  
    val esPar = { v: Int -> v % 2 == 0 }  
    println("Es par 6 = ${esPar(6)}")  
  
    val sumaDivisores = { n: Int ->  
        (1..n / 2).filter { n % it == 0 }.sum()  
    }  
    // Aquí tenemos más de un parámetro de entrada.  
    // Fíjate que no ponemos paréntesis.  
    val sonAmigos = { n1: Int, n2: Int ->  
        sumaDivisores(n1) == n2 && sumaDivisores(n2) == n1  
    }  
    println("Amigos 234, 565 = ${sonAmigos(234, 265)}")  
}
```

3. El **tipo es anónimo** y viene definido por la signatura de la función lambda.

🔴 **Nota:** Solo tiene sentido si vamos a pasar la lambda como **función de orden superior** (HOF).

```
fun main() {  
    // Aquí tiene poco sentido hacer esto.  
    lateinit var esPar: (Int) -> Boolean  
    esPar = { v -> v % 2 == 0 }  
  
    println("Es par 9 = ${esPar(9)}")  
}
```

Definiendo y usando funciones de orden superior HOF

 **Importante:** Este tipo de código y definiciones será muy común encontrarlo en la programación con Android y difiere un poco con la sintaxis de C#, vista en primer curso.

Podemos utilizar un tipo con nombre o no, como ya hemos indicado. Aunque lo más común es no usarlo.

1. Tipado con un SAM interface:

Es más similar a Java o a otros lenguajes como C#, pero no es muy común usarla.

```
fun interface Predicate<T> {
    fun cumple(dato: T): Boolean
}

// El tipo del callback es Predicate<T>
fun<T> muestraPredicado(nombre: String, dato: T,
                      predicadoCallback: Predicate<T>)
= println("${nombre} ${dato} = ${predicadoCallback.cumple(dato)}")

fun esPar(v: Int) = v % 2 == 0

fun main() {
    // Pasamos un identificador de función definido en el mismo ámbito del main
    muestraPredicado("Es par", 6, ::esPar)
    // Pasamos un lambda como parámetro.
    muestraPredicado("Es impar", 6, { v -> v % 2 == 0 })
}
```

2. Tipado con un tipo anónimo:

```
// En este caso el tipo del callback es anónimo y es (T) -> Boolean
// Esto es lo más habitual en Kotlin y lo recomendable.
fun <T> muestraPredicado(
    nombre: String, dato: T,
    predicadoCallback: (T) -> Boolean
) = println("${nombre} ${dato} = ${predicadoCallback(dato)}")

fun main() {
    // Pasamos un lambda como parámetro con la misma signatura del tipo anónimo
    muestraPredicado("Es impar", 6) { v ->
        v % 2 == 0
    }
}
```



Muy importante: Fíjate que al ir la función lambda como último parámetro podemos ponerla fuera de los paréntesis.

- Es válido pero no lo recomendable en Kotlin.

```
muestraPredicado("Es impar", 6, { v -> v % 2 == 0 })
```

- 👉 Esta es la **sintaxis recomendada en Kotlin** y la que vamos usar durante los ejemplos del curso.

```
muestraPredicado("Es impar", 6) { v ->
    v % 2 == 0
}
```

Clausuras

Funcionan de la misma forma que en C#.

```
// Es una HOF porque retorna una función
fun contador() : () -> Int {
    var i : Int = 0;

    // i es una variable clausurada.
    // no ponemos el () -> en la lambda de retorno.
    return { i++ }
}

fun main() {
    var cuenta1 = contador()
    var cuenta2 = contador()

    println("cuenta1 = ${cuenta1()}")
    println("cuenta1 = ${cuenta1()}")
    println("cuenta1 = ${cuenta1()}")
    println("cuenta2 = ${cuenta2()}")
    println("cuenta2 = ${cuenta2()}")
}
```

Map-Filter-Fold

Podemos aplicar los mismos conceptos de primer curso en CSharp.

De hecho el código es muy similar y podremos hacer una analogía fácilmente.

- En **C#** podíamos hacer:

```
double[] notas = [1.0, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7];

// Contamos aquellas notas redondeadas a entero que son mayores o iguales a 5.
int aprobados1 = notas.Select(n => (int)Math.Round(n))
    .Where(n => n >= 5)
    .aggregate(0, (c, n) => c + 1);

// Lo mismo pero usando la función de fold Count a la que le pasamos
// el predicado para filtrar los aprobados.
int aprobados2 = notas.Select(n => (int)Math.Round(n)).Count(n => n >= 5)

// Ordenamos por notas sin repeticiones y tras ordenar de forma
// descendente tomamos el primero.
int notaMayor1 = notas.Select(n => (int)Math.Round(n))
    .Distinct()
    .OrderByDescending(n => n)
    .First();

// Lo mismo pero usando la función de fold Max
int notaMayor2 = notas.Select(n => (int)Math.Round(n)).Max();
```

- El código en **Kotlin** equivalente sería:

```
val notas = listOf(1.0, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7);

val aprobados1 = notas.map { n -> n.roundToInt() }
    .filter { n -> n >= 5 }
    .fold(0) { c, _ -> c + 1 }
val aprobados2 = notas.map { n -> n.roundToInt() }.count { n -> n >= 5 }

val notaMayor1 = notas.map { n -> n.roundToInt() }
    .distinct()
    .sortedByDescending { n -> n }
    .first()
val notaMayor2 = notas.map { n -> n.roundToInt() }.maxOf { n -> n }
```

Creando objetos anónimos de consultas sobre objetos complejos

Ya sabemos un porque que podemos aplicar los conceptos de Map-Filter-Fold a tipos sencillos pero, ¿y si queremos aplicarlos a tipos complejos?. Es más, ¿y si queremos crear objetos anónimos a partir de los resultados de las consultas?

Supongamos la siguiente definición de datos:

```
data class Empleado(val nombre: String, val edad: Int, val ciudad: Ciudad) {
    enum class Ciudad() { Elche, Alicante }
}
object Datos {
    val empleados = listOf(
        Empleado("Xusa", 45, Empleado.Ciudad.Alicante),
        Empleado("Pepe", 54, Empleado.Ciudad.Alicante),
        Empleado("Juanjo", 52, Empleado.Ciudad.Elche),
        Empleado("Vicente", 45, Empleado.Ciudad.Elche))
}
```

Queremos obtener una lista de objetos anónimos con el nombre y la ciudad de los empleados mayores de 45 años sin repeticiones y ordenados por nombre.

```
val res = Datos.empleados
    .filter { e -> e.edad > 45 }      // Filtramos por edad
    .map { e ->                       // Proyectamos lo filtrado a un objeto anónimo
        object {
            val nombre = e.nombre
            val ciudad = e.ciudad
        }
    }
    .distinct()                      // Eliminamos repeticiones
    .sortedBy { d -> d.nombre }      // Ordenamos por nombre
    .toList()                        // Pasamos la secuencia a una lista

// Si quisiéramos retornar la lista
// deberíamos usar una data class en lugar de objetos anónimos.
println(res.joinToString("\n", String.format("%-10s%-10s\n", "Nombre", "Ciudad")){
    d -> String.format("%-10s%-10s", d.nombre, d.ciudad)
})
```

🔴 **Nota:** Nos interesará más usar data class que objetos anónimos cuando queramos hacer consultas sobre objetos complejos y hacer agrupaciones o recuperar los datos de la consulta en una lista.

👉 **Importante::** Los objetos anónimos no definen el toString por defecto como ocurre en C#, por lo que si es necesario se deberá crear un toString dentro de la implementación del objeto:

```
val helloWorld = object {  
    val hello = "Hello"  
    val world = "World"  
    // las expresiones de objeto extienden Any, por lo que se requiere `override` en `to  
    override fun toString() = "$hello $world"  
}
```

Trasformaciones de datos usando map

En la programación con Android nos encontraremos con que tenemos que transformar objetos del **modelo** o dominio en objetos de **presentación** y viceversa. De esta forma evitamos acoplar la vista a los objetos del modelo y viceversa.

Supongamos que guardamos en nuestro modelo de datos una serie de datos de contacto de agenda de la siguiente forma:

```
data class Contacto(  
    val id: Int,  
    val nombre: String,  
    val telefono: String,  
    val listas: EnumSet<Lista>  
) {  
    enum class Lista {  
        Amigos, Trabajo, Familia  
    }  
}
```

Donde **listas** es un conjunto de valores del enumerado **Lista** que indica a qué **lista o listas de contactos** pertenece el registro.

Supongamos una fuente de datos de contactos que nos devuelve una lista de objetos de tipo **Contacto** :

```
object ContactosRepository {  
    val datos = mutableListOf(  
        Contacto(  
            1, "Xusa", "111666666",  
            listas = EnumSet.of(Contacto.Lista.Trabajo, Contacto.Lista.Amigos)  
        ),  
        Contacto(  
            2, "Pepe", "222666666",  
            listas = EnumSet.of(Contacto.Lista.Trabajo, Contacto.Lista.Familia)  
        ),  
        Contacto(  
            3, "Juanjo", "333666666",  
            listas = EnumSet.of(Contacto.Lista.Amigos)  
        ),  
        Contacto(  
            4, "Vicente", "444666666",  
            listas = EnumSet.of(Contacto.Lista.Familia)  
        )  
    )  
}
```

Añadimos un método **Update** para actualizar un contacto en la fuente de datos:

```
object ContactosRepository {  
    ...  
    fun Update(contacto: Contacto) {  
        val index = datos.indexOfFirst { it.id == contacto.id }  
        if (index != -1) {  
            datos[index] = contacto  
        }  
    }  
}
```

En el interfaz de usuario queremos mostrar cada contacto en la lista con un icono que me indique a qué lista de contactos pertenece. Para ello, vamos a crear un tipo en la capa de presentación de tipo **ContactoUiState** con el estado de visualización del interfaz de usuario (UI) que contenga una propiedad booleana para cada una de las posibles listas de contactos...

```
data class ContactoUiState(  
    val id: Int,  
    val nombre: String,  
    val telefono: String,  
    val trabajo: Boolean,  
    val familia: Boolean,  
    val amigos: Boolean  
)
```

Para pasar de un objeto a otro usaremos el método **map** de las colecciones de Kotlin. Pero antes de nada, vamos a crear una función que nos ayude a crear objetos de tipo **ContactoUiState** a partir de objetos de tipo **Contacto**.

Esta función la podemos crear como una **función de extensión** de la clase **Contacto** para evitar acoplamientos y que la clase **Contacto** no tenga que conocer la clase **ContactoUiState**.

```
fun Contacto.toContactoUiState() = ContactoUiState(  
    id = id,  
    nombre = nombre,  
    telefono = telefono,  
    trabajo = listas.contains(Contacto.Lista.Trabajo),  
    familia = listas.contains(Contacto.Lista.Familia),  
    amigos = listas.contains(Contacto.Lista.Amigos)  
)
```

De forma análoga podemos crear una función de extensión para pasar de `ContactoUiState` a `Contacto`.

```
fun ContactoUiState.toContacto() = Contacto(
    id = id,
    nombre = nombre,
    telefono = telefono,
    listas = EnumSet.noneOf(Contacto.Lista::class.java).apply {
        if (trabajo) add(Contacto.Lista.Trabajo)
        if (familia) add(Contacto.Lista.Familia)
        if (amigos) add(Contacto.Lista.Amigos)
    }
)
```

Ahora podemos usar el método `map` para transformar la lista de `Contacto` en una lista de `ContactoUiState` y viceversa de forma sencilla:

```
fun main() {
    // Mostramos los datos de la fuente de datos
    println(ContactosRepository.datos.joinToString(separator = "\n"))

    // Transformamos los datos de la fuente de datos a datos de presentación con map
    val contactosEnUi = ContactosRepository.datos.map { it.toContactoUiState() }.toMutableList()
    println(contactosEnUi.joinToString(separator = "\n"))

    // Modificamos el primer contacto de la lista de presentación
    // para que no pertenezca a la lista de amigos
    contactosEnUi[0] = contactosEnUi[0].copy(amigos = false)

    // Actualizamos los datos de la fuente de datos con los datos de presentación
    // usando map. Aunque lo más normal, será actualizar solo un contacto haciendo
    // ContactosRepository.Update(contactosEnUi[0].toContacto())
    contactosEnUi.map { it.toContacto() }.forEach { ContactosRepository.Update(it) }

    println(ContactosRepository.datos.joinToString(separator = "\n"))
}
```

Haciendo agrupaciones con groupBy

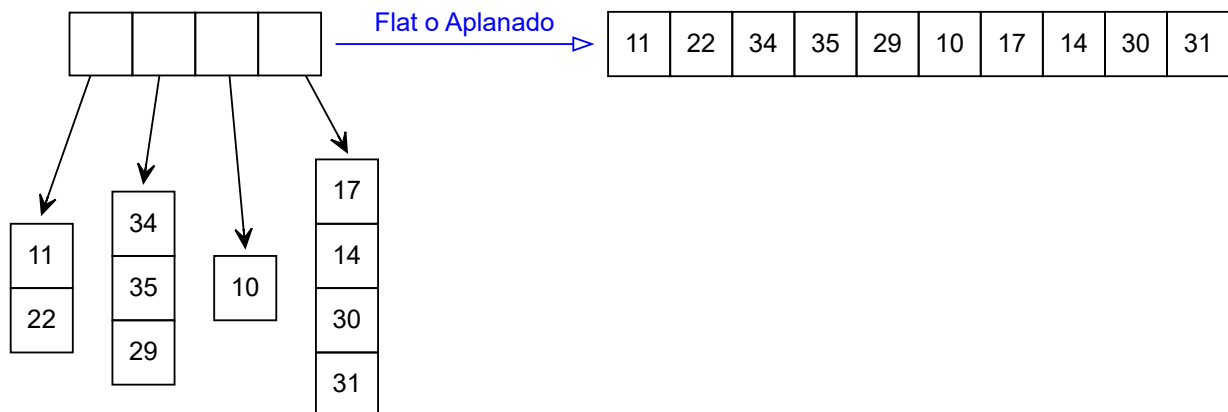
Si quisiéramos mostrar los empleados por ciudad ordenados por nombre podríamos usar el método `groupBy` como en C#.

```
// Fíjate que aunque no es necesario especificar el tipo de la variable
// nosotros lo hemos hecho para ver que la agrupación me
// devuelve un Map<Ciudad, List<Empleado>> donde la clave es la
// propiedad `pr` la que agrupamos.
val empleatsPerCiutat : Map<Empleado.Ciudad, List<Empleado>> =
    Datos.empleados.groupBy { e -> e.ciudad }

var salida : StringBuilder = StringBuilder()
for (eXc in empleatsPerCiutat) {
    salida.append("${eXc.key}:\n")
    eXc.value.sortedBy { e -> e.edad }.forEach {
        e -> salida.append("\t${e}\n")
    }
}
println(salida);
```

Obteniendo secuencias anidadas con flatMap

Supongamos la siguiente de representación donde tenemos el típico array de arrays o tabla dentada. En el fondo, podemos considerarlo como una secuencia de secuencias (sub-secuencias) de enteros:



Si te fijas en el diagrama podemos ver que la operación de flat consiste en generar una nueva secuencia con datos de las sub-secuencias de entrada. El efecto es como si estuviéramos *'aplanando'* la tabla dentada.

En **Kotlin** podemos usar el método `flatMap` para obtener la secuencia de enteros de la tabla dentada. El código sería el siguiente:

```
val jagged = arrayOf(
    arrayOf(11, 22),
    arrayOf(34, 25, 29),
    arrayOf(10),
    arrayOf(17, 14, 30, 31)
)

// flatMap = Array<Array<Int>> → (Array<Int> → List<Int>) → List<Int>
val flat = jagged.flatMap { v -> v.toList() }

println(flat.joinToString(", "))
// Mostrará por pantalla: 11, 22, 34, 25, 29, 10, 17, 14, 30, 31
```

Ejemplo de consulta de datos mapeados a objetos en Kotlin

- Descarga los archivos de este ejemplo de [aquí](#).

Supongamos las siguientes definiciones de a modo de DTOs:

```
data class Libro(  
    val titulo: String,  
    val año: Int,  
    val paginas: Int  
) {  
    override fun toString(): String =  
        "Titulo: ${titulo.padEnd(37)} Año: ${año.toString().padEnd(4)} Páginas: $paginas"  
}  
  
data class Autor(  
    val nombre: String,  
    val nacionalidad: String,  
    val muerte: LocalDate,  
    val libros: List<Libro>  
) {  
    override fun toString(): String =  
        "Nombre: ${nombre.padEnd(37)} Nacionalidad: ${nacionalidad.padEnd(10)} " +  
        "Muerte: ${muerte.format(DateTimeFormatter.ofPattern("dd/MM/yyyy"))}\n" +  
        "Libros:\n\t${libros.joinToString("\n\t")}"  
}
```

y los siguientes datos de prueba:

```
object Datos {  
    val autores: List<Autor> = listOf(  
        Autor(  
            "William Shakespeare",  
            "Inglesa",  
            LocalDate.of(1616, 5, 3),  
            listOf(  
                Libro("Macbeth", 1623, 128),  
                Libro("La tempestad", 1611, 160)  
            )  
        ),  
        Autor(  
            "Miguel de Cervantes",  
            "Española",  
            LocalDate.of(1616, 6, 22),  
            listOf(  
                Libro("Don Quijote de la Mancha", 1605, 1376),  
                Libro("La Galatea", 1585, 664),  
                Libro("Los trabajos de Persiles y Sigismunda", 1617, 888),  
                Libro("Novelas ejemplares", 1613, 1160)  
            )  
        ),  
        Autor(  
            "Fernando de Rojas",  
            "Española",  
            LocalDate.of(1541, 2, 7),  
            listOf(  
                Libro("La Celestina", 1500, 160)  
            )  
        )  
    )  
}  
  
fun separadorDato() =  
    "\n-----\n"
```

Con estos datos de prueba podemos hacer consultas sobre los datos de los autores. Por ejemplo:

1. Si queremos obtener una lista de los autores que han escrito más de un libro podríamos hacer lo siguiente:

```
fun autoresConMasDeUnLibro() {  
    val snapshot = Datos.autores  
        .filter { it.libros.size > 1 }  
    println(snapshot.joinToString(separadorDato()) { it.toString() })  
}
```

2. Si queremos obtener el total de libros escritos por autores españoles podríamos hacer lo siguiente:

```
fun totalLibrosEscritosPorEspañoles() {  
    val totalLibros = Datos.autores  
        // Filtramos por nacionalidad  
        .filter { it.nacionalidad == "Española" }  
        // Obtenemos el número de libros de cada autor  
        .map { it.libros.size }  
        // Sumamos los libros escritos por cada autor  
        .sum()  
    println("Hay $totalLibros escritos por españoles")  
}
```

3. Si queremos obtener una lista de autores agrupados por siglo podríamos hacer lo siguiente:

```
fun autoresAgrupadosPorSiglo() {  
  
    // Creamos un DTO para evitar usar objetos anónimos.  
    data class AutorMuerte(val nombre: String, val muerte: LocalDate)  
    // Nota: Podríamos haber usado un Pair<String, LocalDate> en  
    // lugar de un DTO pero el código sería menos legible.  
  
    val snapshot = Datos.autores  
        // Mapeamos a un objeto tipado AutorMuerte  
        // para hacer la consulta en este ámbito  
        .map { a -> AutorMuerte(nombre = a.nombre, muerte = a.muerte) }  
        // Ordenamos por fecha de muerte  
        .sortedBy { a -> a.muerte }  
        // Agrupamos por siglo obteniendo un Map<Int, List<Autor>>  
        // donde Int es el siglo y List<AutorMuerte> los autores  
        .groupBy { it.muerte.year / 100 + 1 }  
  
    snapshot.forEach { (siglo, autores) ->  
        print(separadorDato())  
        print("Siglo ${siglo}:\n\t")  
        println(autores.joinToString("\n\t") { a ->  
            "${a.nombre} ${a.muerte.format(DateTimeFormatter.ofPattern("dd/MM/yyyy"))}"  
        })  
    }  
}
```

4. Calcular el total de páginas publicadas por William Shakespeare:

```
fun totalPaginasWilliamShakespeare() {  
    val totalPaginas = Datos.autores  
        // Filtramos por nombre  
        .filter { it.nombre == "William Shakespeare" }  
        // De cada autor obtenemos la lista de libros  
        // En este caso solo tendríamos un autor.  
        .flatMap { it.libros }  
        // De cada libro obtenemos el número de páginas  
        .map { it.paginas }  
        // Sumamos las páginas de cada libro  
        .sum()  
    println("William Shakespeare ha escrito $totalPaginas páginas")  
}
```