LAS COLECCIONES

Las colecciones en Kotlin son estructuras de datos dinámicas que permiten almacenar muchos valores del mismo tipo; por tanto, conceptualmente es prácticamente igual que un array.

Los diferentes tipos de colecciones permiten afinar el comportamiento de cada una de ellas, de forma que podremos elegir la que más nos convenga para cada caso.

Existen, por ejemplo, colecciones de datos en las que los elementos están automáticamente ordenados en todo momento (**TreeSet**) u otras que no admiten elementos repetidos (**HashSet** o conjunto). Otras estructuras mantienen el orden de los elementos (**ArrayList** o listas) mientras que en otras esa información no es relevante (**HashMap** o diccionarios).

```
val treeSet: TreeSet<Int> = TreeSet()
val hashSet: HashSet<String> = HashSet()
val arrayList: ArrayList<Double> = ArrayList()
val hashMap: HashMap<String, Int> = HashMap()
```

Se les conoce con el nombre de genéricos, porque puedes indicar el tipo de datos que contendrán al declararlas, pero esto es un concepto general que se aplica a muchos lenguajes de programación y no es específico de la API de Kotlin. Por lo que se les debe pasar un tipo de datos para convertirlos en una colección de un tipo determinado. Así, **en el momento de la declaración tendremos que indicar el tipo de datos que almacenará dicha colección**.

```
val listaLibros: List<Libro> = listOf()

val listaLibros: List<Libro> = listOf(libro1, libro2, libro3)
```

Existe numerosos tipos de datos para almacenar colecciones con diferentes características. En esta sección veremos algunas de las más utilizadas: **ArrayList** y **HashMap**.

Además, existe una serie de interfaces y clases como **List**, **Collection** o **Arrays** de la que extienden o implementan estas estructuras de colección, lo que permite que además existan operaciones que permiten que se puedan hacer operaciones que combinen distintos tipos de colección y otras estructuras de datos como los arrays.

La clase ArrayList

La clase **ArrayList** permite definir arrays dinámicos. Son colecciones de datos dinámicas con el acceso y funcionamiento de un array, puesto que sólo es posible recuperar cada elemento a partir de la posición del mismo.

Un **ArrayList** es una estructura en forma de lista que permite almacenar elementos del mismo tipo (pueden ser incluso objetos); su tamaño va cambiando a medida que se añaden o se eliminan esos elementos.

Como sabemos, el array presenta algunos inconvenientes. Uno de ellos es la necesidad de conocer el tamaño exacto en el momento de su creación. **Una colección, sin embargo, se crea sin que se tenga que especificar el tamaño**; posteriormente se van añadiendo y quitando elementos a medida que se necesitan.

Nos podemos imaginar un **ArrayList** como un conjunto de celdas o cajoncitos donde se guardan los valores, exactamente igual que un array convencional. En la práctica será más fácil trabajar con un **ArrayList**.

En Kotlin, tenemos los siguientes tipos de Listas:

- **Listas No Mutables (List)**: Una lista no mutable se representa por la interfaz **List**. La lista no mutable es de solo lectura; es decir, una vez que se crea, no se pueden realizar operaciones de modificación, como agregar, eliminar o cambiar elementos. Puedes acceder a los elementos y realizar operaciones de lectura, pero no se pueden modificar.

```
val listaNoMutable: List<String> = listOf("Manzana", "Banana", "Cereza")
```

- **Listas Mutables (MutableList):** Una lista mutable se representa por la interfaz **MutableList**. A diferencia de las listas no mutables, las listas mutables pueden ser modificadas después de su creación. Puedes agregar, eliminar o cambiar elementos en una lista mutable.

```
val listaMutable: MutableList<String> = mutableListOf("Manzana", "Banana")
listaMutable.add("Dátil") // Se agrega un nuevo elemento

listaMutable.removeAt(1)  // Se elimina el elemento en la posición 1
```

ArrayList es una clase específica de Kotlin que implementa la interfaz **MutableList**.

```
val lista: MutableList<Int> = ArrayList()
lista.add(18)
lista.add(22)
lista.add(-30)

println("N° de elementos: ${lista.size}")
println("El elemento que hay en la posicion 1 es: $lista[1]")
```

Entre las operaciones que se pueden realizar con un **ArrayList** las más comunes son las siguientes:

Añadir un elemento al final de la lista

```
val listaLibros = ArrayList<Libro>()
val libro = Libro(/* ... */)
listaLibros.add(libro)
```

NOTA: Existe el método **add(indice, elemento)**, que inserta un elemento en una posición determinada, desplazando el resto de elementos hacia la derecha.

```
val listaLibros = ArrayList<String>()
listaLibros.add("Libro 1")
listaLibros.add("Libro 2")
listaLibros.add("Libro 3")

// Insertar un nuevo libro en la posición 1
listaLibros.add(1, "Libro Nuevo")

// Resultado después de la inserción: //
["Libro 1", "Libro Nuevo", "Libro 2", "Libro 3"]
```

Añadir toda una colección al final

```
val listaLibros = ArrayList<Libro>()
val otraListaLibros = ArrayList<Libro>()
// ...

listaLibros.addAll(otraListaLibros)
```

Obtener un elemento indicando la posición

```
val unLibro: Libro = listaLibros[4]

val libroPos: Libro =listaLibros.get(4)
```

Eliminar un elemento indicando la posición

```
val libroEliminado: Libro? = listaLibros.removeAt(4)
```

Es importante mencionar que **removeAt** devuelve el elemento eliminado y también lo elimina de la lista. La declaración del tipo de la variable libroEliminado incluye el signo de interrogación (?) para indicar que el valor puede ser nulo si el índice está fuera de los límites de la lista. Si la eliminación es exitosa, libroEliminado contendrá el valor eliminado; de lo contrario, será nulo.

Eliminar la primera ocurrencia del elemento indicado

Devuelve true si la lista contenía el elemento que se especifica y fue eliminado, y false en caso contrario. Si la lista contiene objetos de clases propias, solo funcionará correctamente si la clase implementa el método *equals*.

```
val booRemoved: Boolean = listaLibros.remove(libro)
```

Eliminar todos los elementos, dejando la lista vacía

```
listaLibros.clear()
```

Obtener el tamaño de la colección (número de elementos)

```
val tamano: Int = listaLibros.size
```

Obtener un array con los elementos de la colección

```
val libros: Array<Libro> = listaLibros.toTypedArray()
```

Comprobar si existe un elemento

Devuelve true si la lista contiene el elemento que se especifica y false en caso contrario. Si la lista contiene objetos de clases propias, solo funcionará correctamente si la clase implementa el método *equals*.

```
val booContieneElemento: Boolean = listaLibros.contains(libro)
```

Devuelve la posición de la primera ocurrencia del elemento que se indica entre paréntesis.

```
val posicion: Int = listaLibros.indexOf(libro)
```

Esta línea de código obtiene la posición del elemento libro en la lista listaLibros. Si el elemento no está presente en la lista, **indexOf** devuelve -1. Si la lista contiene objetos de clases propias, solo funcionará correctamente si la clase implementa el método *equals.*

Comprobar si la lista está vacía

Devuelve true si la lista está vacía y false en caso de tener algún elemento.

```
val booEsVacia: Boolean = listaLibros.isEmpty()
```

Sobreescribir elemento

Machaca el elemento que se encuentra en una determinada posición con el elemento que se pasa como parámetro. Devuelve el elemento que se encontraba antes en dicha posición.

```
val libroAnterior: Libro = listaLibros.set(1, nuevoLibro)
```

A continuación, se muestra un **ejemplo** en el que se puede ver cómo se declara un **ArrayList** y cómo se insertan y se leen todos los elementos de una lista a la manera tradicional, con un bucle for:

Observa que al definir un objeto de la clase **ArrayList** hay que indicar el tipo de dato que se almacenará en las celdas de esa lista. Para ello se utilizan los caracteres < y >.

```
fun main() {
    // Declaración e inicialización del ArrayList
    val listaColores: MutableList<String> = ArrayList()

    // Insertar elementos en la lista
    listaColores.add("Rojo")
    listaColores.add("Verde")
    listaColores.add("Azul")
    listaColores.add("Amarillo")

    // Mostrar los elementos de la lista
    println("Colores en la lista:")
    for (color in listaColores) {
        println(color)
    }
}
```

Salida que se generará por consola:

```
Colores en la lista:
Rojo
Verde
Azul
Amarillo
```

Veamos ahora cómo eliminar elementos de un **ArrayList**. Se utiliza el método **remove**() y se puede pasar como parámetro el índice del elemento que se quiere eliminar, o bien el valor del elemento.

O sea, removeAt(2) elimina el elemento que se encuentra en la posición 2 de la lista, mientras que remove("Verde") elimina el valor "Verde".

Es importante destacar que el **ArrayList** se reestructura de forma automática después del borrado de cualquiera de sus elementos.

```
fun main() {
    // Declaración e inicialización del ArrayList
    val listaColores: MutableList<String> = ArrayList()

    // Añadir elementos a la lista
    listaColores.add("Rojo")
    listaColores.add("Verde")
    listaColores.add("Azul")
    listaColores.add("Amarillo")
    println("Colores:")
    mostrarLista(listaColores)


    if (listaColores.contains("Rojo")){
        println("El color Rojo está en la lista");
    }

    // Eliminar el color 'Verde'
    val eliminado: Boolean = listaColores.remove("Verde")
    println("¿Se eliminó el color 'Verde'? $eliminado")
    println("Colores después de eliminar 'Verde':")
    mostrarLista(listaColores)

    // Obtener el color en la posición 2
    val colorEnPosicion2: String? = listaColores.getOrNull(2)
    println("Color en la posición 2: $colorEnPosicion2")

    // Eliminar el color de la posición 2
    val colorEliminado: String = listaColores.removeAt(2)
    println("Se eliminó el color $colorEliminado")
    println("Colores después de eliminar:")
    mostrarLista(listaColores)

    // Limpiar la lista
    listaColores.clear()
    println("Colores después de limpiar la lista:")
    mostrarLista(listaColores)
}

fun mostrarLista(lista: MutableList<String>) {
    for (color in lista) {
        println(color)
    }
}
```

Ordenación de un ArrayList

Los elementos de una lista se pueden ordenar con el método **sort** de la interfaz **MutableList**, que internamente lo que hace es llamar a java.util.Collections.**sort**(lista).

Si queremos ordenar una lista de objetos. La clase de elementos que queremos ordenar, deberá implementar el interfaz **Comparable,** así como definir el método compareTo.

El método compareTo toma un argumento (el objeto con el que se está comparando) y devuelve un entero. La convención general es la siguiente:

- Devolver un número negativo si el objeto actual es menor que el objeto de comparación.
- Devolver cero si son iguales.
- Devolver un número positivo si el objeto actual es mayor que el objeto de comparación.

Las clases String, Integer, Double, etc. ya tienen implementado su propio método compareTo.

En este ejemplo, utilizamos el método **sort** directamente sobre el **ArrayList** de palabras (listaPalabras). Como la clase **String** ya implementa el interfaz *Comparable* en Kotlin, al igual que en Java, no necesitas definir tu propio método ***compareTo*** para la ordenación.

```
// Ejemplo de ordenación de un ArrayList de String
fun main() {
    val listaPalabras: ArrayList<String> = arrayListOf("Zorro", "León", "Perro",
"Gato", "Elefante")

    // Antes de ordenar
    println("Lista antes de ordenar:")
    mostrarLista(listaPalabras)

    // Ordenar la lista de manera ascendente
    listaPalabras.sort()

    // Después de ordenar
    println("Lista después de ordenar de manera ascendente:")
    mostrarLista(listaPalabras)
}

fun mostrarLista(lista: ArrayList<String>) {
    for (elemento in lista) {
        println(elemento)
    }
}
```

Según lo que se indicó anteriormente, si queremos ordenar una lista de objetos, en la clase de elementos que queremos ordenar, deberá implementarse el interfaz **Comparable,** así como definir el método compareTo, el cual permite definir el criterio de ordenación natural para una clase.

Por tanto, la clase **Libro** implementa **Comparable<Libro>,** lo que significa que estás proporcionando una implementación personalizada de cómo se deben comparar dos objetos **Libro**.
El método compareTo de **Libro** se sobrescribe, por eso aparece la palabra override, para comparar los libros por su título, utilizando el método compareTo de la clase **String**, que ya compara cadenas lexicográficamente.

```
class Libro(val titulo: String, val autor: String, val añoPublicacion: Int) : Comparable<Libro> {
    override fun compareTo(other: Libro): Int {
        // Comparación por título
        return this.titulo.compareTo(other.titulo)
    }
}

class OrdenadorLibros {
    companion object {
        fun ordenarPorTitulo(listaLibros: ArrayList<Libro>) {
                listaLibros.sort()
        }

        fun mostrarLista(lista: ArrayList<Libro>) {
                for (libro in lista) {
                        println(libro.titulo)
                }
        }
    }
}

fun main() {
    // Crear una lista de libros
    val listaLibros: ArrayList<Libro> = arrayListOf(
        Libro("Cien años de soledad", "Gabriel García Márquez", 1967),
        Libro("1984", "George Orwell", 1949),
        Libro("To Kill a Mockingbird", "Harper Lee", 1960)
    )

    // Antes de ordenar
    println("Lista de libros antes de ordenar:")
    OrdenadorLibros.mostrarLista(listaLibros)

    // Ordenar la lista de libros por título
    OrdenadorLibros.ordenarPorTitulo(listaLibros)

    // Después de ordenar
    println("Lista de libros después de ordenar por título:")
    OrdenadorLibros.mostrarLista(listaLibros)
}
```

Otra forma de ordenar elementos de una lista es usar la función sortWith en Kotlin la cual ordena la lista, de acuerdo al comparador indicado.

Para utilizar esta función, en primer lugar, deberemos crear el "comparador", lo cual consiste en hacer una clase que implemente el interfaz **Comparator** para el tipo de datos que se quieren ordenar, así como definir el método ***compare***, el cual compara los dos objetos pasados por parámetro, y permite definir como se quieren ordenar los objetos.

```
class ComparadorLibros: Comparator<Libro> {
    override fun compare(o1: Libro?, o2: Libro?): Int {
        if(o1 == null || o2 == null)
            return 0
        return o1.titulo.compareTo(o2.titulo)
    }
}
```

El método compare se sobrescribe, por eso aparece la palabra *override*, para comparar los libros por su título, utilizando el método compareTo de la clase **String**, que ya compara cadenas lexicográficamente.

A continuación, vemos un ejemplo de cómo podemos usar esta forma de ordenar:

```
val comparador = ComparadorLibros()
listaLibros.sortWith(comparador)
```

Esto es lo que hace internamente la función compareBy:

```
val ordenComparador: Comparator<Libro> = compareBy({ it.titulo })
listaLibros.sortWith(ordenComparador)
```

La clase HashMap

Es una tabla hash que implementa el interfaz **Map** de Java, lo que le convierte en una estructura de colección que almacena **objetos asociados a una clave**.

La clase **HashMap** funciona como un diccionario, contiene una serie de elementos que son las entradas que a su vez están formadas por un par (clave, valor). La clave (key) permite acceder al valor. **No puede haber claves duplicadas**.

Este tipo de colecciones, a diferencia de los anteriores, no garantiza el orden de los elementos, puesto que éstos se pueden obtener solamente utilizando la clave que se asoció al mismo en el momento de añadirlo.

```
// Declaración de un diccionario en Kotlin
val mapa: HashMap<Int, String> = HashMap()
// Añadir elementos al diccionario
mapa[1] = "Valor 1"
mapa[2] = "Valor 2"
mapa[3] = "Valor 3"

println(mapa)

// Acceder a un valor utilizando la clave
val valorParaClave2: String? = mapa[2]
println("Valor asociado a la clave 2: $valorParaClave2")
```

```
fun main() {
    // Declaración de un diccionario en Kotlin con inferencia de tipo
    val mapa = hashMapOf(
        1 to "Manzana",
        2 to "Banana",
        3 to "Cereza",
        4 to "Dátil"
    )

    // Acceder a un valor utilizando la clave
    val frutaParaClave2: String? = mapa[2]
    println("Fruta asociada a la clave 2: $frutaParaClave2")

    // Mostrar todos los elementos del diccionario
    println("Contenido del diccionario:")
    for ((clave, valor) in mapa) {
        println("Clave: $clave, Valor: $valor")
    }
}
```

Entre las operaciones que se pueden realizar con un **HashMap** las más comunes son las siguientes:

Añadir un elemento (pareja clave-valor)

```
val libros: HashMap<String, Libro> = hashMapOf()
val libro = Libro("titulo", "autor", 2024)

//dos formas de insertar
libros.put(libro.titulo, libro)

libros += libro.titulo to libro
```

Obtener un elemento

```
val unLibro: Libro? = libros[“titulo”]

val libro: Libro? = libros.get(“titulo”)
```

Comprobar si existe un elemento (a través de su clave)

Devuelve true si el diccionario contiene la clave indicada y false en caso contrario.

```
val booEncontrado: Boolean = libros.containsKey(tituloLibro)
```

Eliminar todos los elementos

```
libros.clear()
```

Eliminar un elemento

```
val previous: Libro? = libros.remove(“titulo”)
```

Elimina la entrada correspondiente a la clave “titulo” del mapa libros y devuelve el valor asociado a esa clave antes de la eliminación. La variable *previous* es de tipo Libro? para reflejar que el valor puede ser nulo si la clave no estaba presente en el mapa.

Obtener una colección con todas las claves

Set: son considerados listas que **no garantizan un orden en concreto**, por ello, no puedes tener certeza del orden de los objetos que allí se almacenan

```
val losTitulos: Set<String> = libros.keys
```

Obtener una colección con todos los valores

```
val losLibros: Collection<Libro> = libros.values
```

Obtener el número de elementos del HashMap

```
val tamanio: Int = libros.size
```

Veamos un **ejemplo** completo, de cómo insertar una entrada en el diccionario que tiene como clave un entero y valor una cadena de caracteres. Para añadir una entrada, siempre hay que indicar el par: clave/valor.

```
class Libro(val titulo: String, val autor: String, val annio: Int){
 override fun toString(): String {
    return "Libro(titulo='$titulo', autor='$autor', añoPublicacion=$añoPublicacion)"
 }

}

fun main() {
    // Crear un diccionario (HashMap) con clave entera y valor String
    val libros: HashMap<String, Libro> = hashMapOf()

    // Crear un libro
    val nuevoLibro = Libro("Nuevo Libro", "Autor Desconocido", 2024)

    // Añadir una entrada al diccionario con la nueva clave y valor
    libros.put(nuevoLibro.titulo, nuevoLibro)

    // Crear un Set con todas las claves del diccionario
    val clavesLibros: Set<String> = libros.keys

        // Imprimir las claves del diccionario
    println("Claves del diccionario:")
    for (clave in clavesLibros) {
        println(clave)
    }

    // Crear una Collection con todos los valores del diccionario
    val valoresLibros: Collection<Libro> = libros.values

   // Imprimir los valores del diccionario
    println("Valores del diccionario:")
    for (valor in valoresLibros) {
        println(valor)
    }


    // Verificar si el diccionario contiene una clave específica
    val contieneClave: Boolean = libros.containsKey(nuevoLibro.titulo)
    println("¿Contiene la clave del nuevo libro?: $contieneClave")

    // Verificar si el diccionario contiene un valor específico
    val contieneValor: Boolean = libros.containsValue(nuevoLibro)
    println("¿Contiene el valor del nuevo libro?: $contieneValor")

    // Verificar si el diccionario está vacío
    val estaVacio: Boolean = libros.isEmpty()
    println("¿El diccionario está vacío?: $estaVacio")

    // Eliminar una entrada del diccionario
    val libroEliminado: Libro? = libros.remove(nuevoLibro.titulo)
    println("Libro eliminado del diccionario: $libroEliminado")
```

Las Pilas

Las Pilas son colecciones de datos donde éstos se colocan *apilándolos* y sólo puede ser retirado el elemento que se encuentra encima de la pila. Siguen el principio LIFO (Last In First Out) donde el último elemento en llegar a la colección es el primero en salir de la misma.

Las Colas

Las colas son colecciones donde los elementos son gestionados según el principio FIFO (First In First Out). Funcionan, básicamente, como la *cola del cine*. El primer elemento que llega a la cola será el primero en salir. Si un elemento se añade a la cola, tendrá que esperar a que salgan todos los que le preceden para salir él.

En Kotlin, la clase **LinkedList** implementa el interfaz Queue y proporciona una cola FIFO a través de los métodos add(Object) para añadir y poll() para retirar el primer elemento.

CLASES DE DATOS

Las data classes en Kotlin, son utilizadas para almacenar datos. Un ejemplo de cómo definir este tipo de clases es:

```
data class Usuario(var nombre: String, var edad: Int)
```

Hay que poner el modificador **data** delante de **class**, y es obligatorio definir un constructor primario con al menos una propiedad definida como var o val.

Una de las características de las data classes, es que automáticamente se crearán las siguientes funciones, con las propiedades definidas dentro del constructor primario, es decir, si una propiedad es declarada en el cuerpo de la clase, en lugar de en el constructor primario, no estará en las funciones que indicamos a continuación.

```
data class Usuario(var nombre: String, var edad: Int){
      var esCasado: Boolean = false
}
```

- equals(): Compara dos objetos.
- hashCode(): Código hash de un objeto usado por el método anterior
- toString(): Convierte objeto a string
- copy: La función copy() permite hacer una copia del objeto, y además, la implementación que hace de la función, permite cambiar algunas de sus propiedades, manteniendo sin cambios el resto.

  ```
  val usuario= Usuario("Luis", 18)
  val usuarioCopia = usuario.copy(edad = 4)
  ```

- componentN(): Crea internamente una función por cada propiedad del constructor primario, donde N es la posición de la propiedad en la declaración del constructor primario.

  ```
  val usuario= Usuario("Luis", 18)
  val nombre = usuario.component1()
  val edad = usuario.component2()
  ```

  Esto permite que al usar declaraciones de variables desestructuradas, internamente se usen estas funciones componentN()

  ```
  val (nombre, edad) = usuario
  ```

  NOTA: Desestructurar es el procedimiento por el cual podremos extraer múltiples valores que se encuentran almacenados en objetos. De esta forma podremos acceder a cada uno de los valores de una clase de datos utilizando una sola asignación.

  Si algún valor no nos interesa usamos _ : `val (_, edad) = usuario`

  Otra de las utilizaciones más comunes de la desestructuración es para descomponer objetos almacenados en un Map por su clave y valor.

  ```
  val mapa = HashMap<String, Int>()
  for ((clave, valor) in mapa) { … }
  ```

Existe la clase de datos **Pair**, que representa un par de valores, y nos puede ser de ayuda por ejemplo cuando trabajamos con matrices, por ejemplo, para guardar el valor de la coordenada x e y.

A continuación, se muestra la declaración de esta clase:

```
public data class Pair<out A, out B>(
    public val first: A,
    public val second: B
) : Serializable
```

Y un ejemplo de uso:

```
val pair = Pair(first = 1, second = "value")
val (first, second) = pair
```

O si por ejemplo queremos acceder a un único valor:

```
val first = pair.first
```