Taller Estructuras de Datos en Kotlin

- 1. Introducción a las estructuras de datos en Kotlin
  - a. ¿Qué son las estructuras de datos y para qué se utilizan?

    Las estructuras de datos son formas de organizar y almacenar datos de manera eficiente, se utiliza para gestionar grandes cantidades de datos, almacenar y recuperar información de manera rápida y es una herramienta fundamental para la manipulacion de datos.
  - b. Ventajas de utilizar estructuras de datos en Kotlin

    el uso de estructuras de datos en Kotlin puede mejorar la eficiencia, organización y flexibilidad de un programa, al tiempo que facilita la reutilización de código y el mantenimiento a largo plazo.
  - c. Diferencias entre las estructuras de datos en Kotlin y c#

    Kotlin y C# son lenguajes de programación que ofrecen una amplia variedad de estructuras de datos, aunque presentan algunas diferencias entre tipos de datos, colecciones, propiedades y campos, tipos anulables y expresiones lambda.
    - Tipos de datos

      en kotlin esta el tipo de dato "Any” que se utliza para representar cualquier tipo de objeto no esta disponible para c# .En c# no existe un tipo de dato que represente un objeto pero nos ofrece un tipo de dato "object” que sirve para el mismo fin.
    - Estructura de datos:

      Ambos lenguajes proporcionan estructuras de datos como listas, mapas, conjuntos, entre otros. Kotlin ofrece unas estructuras disponibles como "Sequence" y "Range", que no estan en c#, por otro lado c# ofrece la clase "Queue” que no esta en kotlin.
    - Propiedades y campos:

      En Kotlin, las propiedades se definen utilizando la palabra clave "val" para propiedades de solo lectura y "var" para propiedades que pueden ser modificadas. En C#, las propiedades se definen utilizando las palabras clave "get" y "set". Además, en C# se pueden definir campos públicos, mientras que en Kotlin todas las propiedades deben tener un getter y un setter.

▼ Tipos anulables:

En Kotlin, todas las variables son anulables por defecto, lo que significa que pueden ser nulas. En C#, todas las variables son no anulables por defecto, pero se pueden hacer anulables mediante el uso del operador "?".

▼

▼ 2. Arreglos en Kotlin

▼ ¿Qué es un arreglo?

un arreglo es una estructura de datos que permite almacenar diferentes datos de un mismo tipo en una única variable. Se define utilizando la palabra clave "arrayOf" seguida de los elementos a almacenar. Los arreglos son inmutables por defecto, pero también existe la variante "Array" que se puede modificar después de su creación.

▼ Creación de arreglos en Kotlin

▼ Funcion Arrayof

Para declarar un **arreglo en Kotlin** se usa la función (arrayof) que de igual forma se puede indicar que tipo de dato es. Despues se indican los elementos del arreglo como argumentos a la función.

```
// Definir un arreglo con arrayOf
val arreglo = arrayOf(elemento1, elemento2, ...)
val nombres = arrayOf("Luis", "María José", "Fernando")
```

▼ Constructor Array

Utilizando el constructor (datos que se pueden cambiar) "Array", que permite crear un arreglo con el tamaño y el tipo de datos especificados .

```
//(primero va la cantidad de datos, despues indicaremos que en cada indice
//guardamos cadenas de textos (lambam))
var arreglo = Array(5, {i -> "hola"})
```

▼ Accediendo a los elementos de un arreglo

para acceder a los elementos de un arreglo se utiliza el operador de indece que se utiliza los corchetes junto con el indice de elemento.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
// indece del elemento es [0] osea que mostrara 1
val primerNumero = numeros[0]
// o si no queremos usar los corchetes utilizamos la funcion get()
val primerNumero = numeros.get(0)
```

▼ Modificando los elementos de un arreglo

Para modificar un elemento de un array se accede al indice o posicion del elemento que deseas modificar seguido del nuevo valor que deseas actualizar.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
// accedemos al indice y damos el nuevo valor
numeros[1] = 10

// otra forma es llamando a la funcion set()
// get(indice del elemento, valor nuevo)
numeros.set(3, 13)
```

▼ Recorriendo un arreglo

se puede recorrer los elemntos con diferentes tipos de bucles o funciones como el bucle “for” y el bucle “for each” y funciones como el “forEach” y “map”

Para iterar a lo largo de un arreglo con el bucle `for`, debemos usar como base los índices de sus elementos.

Esto significa, que usamos en la declaración de variable el índice y luego expresamos su existencia sobre la propiedad rango entero `indices`.

```
// para recorrer con un bucle for, debemos usar como base
// los índices de sus elementos.Esto quiere decir que usamos en la declaración
// de variable el índice y luego expresamos su existencia sobre la propiedad rango
// entero indices.
val numeros = arrayOf(1, 2, 3, 4, 5)
for (i in numeros.indices) {
    println(numeros[i])
}

// Llamamos a la funcion forEach y si le pasa como argumento una funcion lambda

val numeros = arrayOf(1, 2, 3, 4, 5)

numeros.forEach { numero ->
    println(numero)
}
```

▼ Funciones útiles para trabajar con arreglos en Kotlin

1. size: devuelve el tamaño del arreglo.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val tamano = numeros.size // 5
```

2. forEach: recorre todos los elementos del arreglo y realiza una acción para cada elemento.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
numeros.forEach { numero -> println(numero) }
```

3. map: devuelve un nuevo arreglo con los elementos transformados por una función.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val numerosDobles = numeros.map { numero -> numero * 2 }
```

4. filter: devuelve un nuevo arreglo con los elementos que cumplen con una condición.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val numerosPares = numeros.filter { numero -> numero % 2 == 0 }
```

5. reduce: reduce el arreglo a un solo valor utilizando una función.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val sumaTotal = numeros.reduce { acumulado, numero -> acumulado + numero }
```

6. sorted: devuelve un nuevo arreglo ordenado.

```
val numeros = arrayOf(5, 2, 4, 1, 3)
val numerosOrdenados = numeros.sorted()
```

7. **distinct**: devuelve un nuevo arreglo con los elementos únicos.

```
val numeros = arrayOf(1, 2, 3, 3, 4, 4, 5)
val numerosUnicos = numeros.distinct()
```

8. indexOf: devuelve el índice del primer elemento que coincide con un valor.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val indice = numeros.indexOf(3) // 2
```

9. copyOf: devuelve una copia del arreglo original.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val numerosCopia = numeros.copyOf()
```

▼ 3. Listas en Kotlin

▼ ¿Qué es una lista?

Las listas son colección ordenada de elementos que pueden contener duplicados. Se puede acceder a los elementos de la lista por su índice y se puede agregar, eliminar y modificar elementos de la lista. Hay dos tipos principales de listas en Kotlin: la lista inmutable (List) y la lista mutable (**MutableList**). La lista inmutable es de sólo lectura, mientras que la lista mutable se puede modificar después de que se crea.

▼ Creación de listas en Kotlin

```
// crea una lista vacia y se puede agregar el tipo de dato
val listaVacia = emptyList<String>()

// crea una lista de elementos especificos
// la funcion list es para las listas inmutables
val lista = listOf("manzana", "naranja", "plátano")
// la funcion mutableList es para las listas mutables
  val listaMutable = mutableListOf("manzana", "naranja", "plátano")

// Crea una lista utilizando un rango de valores
val numeros = (1..10).toList() // crea una lista de números del 1 al 10

// Crea una lista utilizando una expresión lambda
val numeros = List(10) { it * 2 } // crea una lista de números pares del 2 al 20
```

▼ Accediendo a los elementos de una lista

se pueden acceder a una elemento de una lista por su indece (posicion) por ejemplo:

```
val frutas = listOf("manzana", "naranja", "plátano")

// Acceder al primer elemento de la lista
val primeraFruta = frutas[0] // "manzana"

// Acceder al segundo elemento de la lista
val segundaFruta = frutas[1] // "naranja"

// Acceder al último elemento de la lista
val ultimaFruta = frutas[frutas.size - 1] // "plátano"

// la funcion last accede al ultimo elemento de la lista
val ultimaFruta = frutas.last() // "plátano"

// la funcion get(indice) accede a los elementos de la lista con su indice
val fruta = frutas.get(3) // devuelve null
```

▼ Modificando los elementos de una lista

En Kotlin, se puede modificar los elementos de una lista si se utiliza una lista mutable (**MutableList**). Si se utiliza una lista inmutable (List), los elementos de la lista no se pueden modificar después de que se crea.

```
val numeros = mutableListOf(1, 2, 3, 4, 5)

// Para modificar se tiene en cuenta el indice que se va a modificar y
// el valor nuevo

// Modificar el primer elemento de la lista
numeros[0] = 6 // ahora la lista es [6, 2, 3, 4, 5]

// Agregar un nuevo elemento al final de la lista
numeros.add(8) // ahora la lista es [6, 3, 7, 4, 5, 8]

// Eliminar un elemento de la lista
numeros.removeAt(0) // ahora la lista es [3, 7, 4, 5, 8]
```

▼ Recorriendo una lista

1. Utilizando un bucle for para iterar sobre los elementos de la lista.

```
val numeros = listOf(1, 2, 3, 4, 5)

for (numero in numeros) {
    println(numero)
}
```

2. Utilizando la propiedad indeces de la lista y un bucle for para acceder a cada elemento de la lista mediante su índice.

```
val numeros = listOf(1, 2, 3, 4, 5)

for (i in numeros.indices) {
    println(numeros[i])
}
```

3. Utilizando la función forEach() para ejecutar una función para cada elemento de la lista.

```
val numeros = listOf(1, 2, 3, 4, 5)

numeros.forEach {
```

```
    println(it)
}
```

4. Utilizando la función **forEachIndexed** para ejecutar una función para cada elemento de la lista y acceder tanto al índice como al valor de cada elemento.

```
val numeros = listOf(1, 2, 3, 4, 5)

numeros.forEachIndexed { index, value ->
    println("El elemento en la posición $index es $value")
}
```

▼ Funciones útiles para trabajar con listas en Kotlin

1. filter( ): devuelve una nueva lista que contiene solo los elementos que cumplen con una determinada condición.

```
val numeros = listOf(1, 2, 3, 4, 5)

val numerosPares = numeros.filter { it % 2 == 0 }

// Output: [2, 4]
```

2. map( ) : devuelve una nueva lista que contiene los elementos resultantes de aplicar una determinada operación a cada elemento de la lista original.

```
val numeros = listOf(1, 2, 3, 4, 5)

val cuadrados = numeros.map { it * it }

// Output: [1, 4, 9, 16, 25]
```

3. reduce( ) : combina los elementos de la lista utilizando una operación específica y devuelve un único resultado.

```
val numeros = listOf(1, 2, 3, 4, 5)

val suma = numeros.reduce { acc, numero -> acc + numero }

// Output: 15
```

4. fold( ) : combina los elementos de la lista utilizando una operación específica y devuelve un único resultado, pero también permite especificar un valor inicial para la operación.

```
val numeros = listOf(1, 2, 3, 4, 5)

val sumaInicial = 10

val suma = numeros.fold(sumaInicial) { acc, numero -> acc + numero }

// Output: 25
```

▼ 4. Conjuntos en Kotlin

▼ ¿Qué es un conjunto?

Un conjunto es una colección de elementos sin orden y sin elementos duplicados. Son útiles en situaciones en las que se necesita almacenar un conjunto de elementos únicos y no importa el orden en que se almacenen.

▼ Creación de conjuntos en Kotlin

1. Usando la función setOf() para crear un conjunto inmutable con los elementos especificados:

```
val set1 = setOf(1, 2, 3, 4, 5)
```

2. Usando la función mutableSetOf() para crear un conjunto mutable con los elementos especificados:

```
val set2 = mutableSetOf("apple", "banana", "orange")
```

También se pueden crear conjuntos vacíos usando la función setOf()o mutableSetOf(), sin argumentos.

```
val emptySet1 = setOf<Int>()
val emptySet2 = mutableSetOf<String>()
```

▼ Accediendo a los elementos de un conjunto

En los conjuntos no se puede acceder al elmento por el indice como las listas o los array, pero hay un metodo contains() para verificar si un elemento está presente en el conjunto que devuelve un valor Booleano.

```
val mySet = setOf("apple", "banana", "orange")
val containsBanana = mySet.contains("banana")
val containsPear = mySet.contains("pear")
```

```
println(containsBanana) // Output: true
println(containsPear) // Output: false
```

Para acceder a los elementos de un conjunto a través de la iteración, puedes utilizar un bucle for-each o un método forEach. Por ejemplo:

```
val mySet = setOf("apple", "banana", "orange")
for (element in mySet) {
    println(element)
}
// Output:
// apple
// banana
// orange

mySet.forEach { element ->
    println(element)
}
// Output:
// apple
// banana
// orange
```

▼ Modificando los elementos de un conjunto

los conjuntos set son estructuras inmutables por lo que no se puede modificar los elementos, por eso para modificar debes crear otro conjunto con los elementos actualizados.

```
// Crear un nuevo conjunto con elementos actualizados
val originalSet = setOf(1, 2, 3)
// En este ejemplo usamos map se utiliza para multiplicar cada elemento original * 2
// y toSet() para crear el nuevo conjunto
val updatedSet = originalSet.map { it * 2 }.toSet() // [2, 4, 6]

// Crear un nuevo conjunto sin un elemento específico
val originalSet = setOf("apple", "banana", "orange")
// usamos filter para crear un nuevo conjunto con los elementos
// originales y eliminamos un elemento.
// y toSet() para crear el conjunto nuevo con su cambio
val updatedSet = originalSet.filter { it != "banana" }.toSet() // ["apple", "orange"]
```

▼ Recorriendo un conjunto

Para recorrer un conjunto (Set) utilizamos un ciclo for o forEach(). El ciclo for te permite recorrer cada elemento del conjunto de manera secuencial, mientras que forEach() es una función de orden superior que toma una función lambda como argumento y la aplica a cada elemento del conjunto.

```
// Recorriendo un conjunto con ciclo for
val set = setOf("apple", "banana", "orange")

for (item in set) {
    println(item)
}
// Output: "apple", "banana", "orange"

// Recorriendo un conjunto con forEach()
val set = setOf("apple", "banana", "orange")

set.forEach { item ->
    println(item)
}
// Output: "apple", "banana", "orange"
```

▼ Funciones útiles para trabajar con conjuntos en Kotlin

1. union() : esta función crea un nuevo conjunto que contiene todos los elementos de dos conjuntos. Los elementos duplicados se eliminan automáticamente.

```
val set1 = setOf("apple", "banana", "orange")
val set2 = setOf("banana", "pear")

val unionSet = set1.union(set2)
println(unionSet) // Output: ["apple", "banana", "orange", "pear"]
```

2. **intersect()**: esta función crea un nuevo conjunto que contiene solo los elementos que se encuentran en ambos conjuntos.

```
val set1 = setOf("apple", "banana", "orange")
val set2 = setOf("banana", "pear")

val intersectSet = set1.intersect(set2)
println(intersectSet) // Output: ["banana"]
```

3. **subtrasct()**: esta función crea un nuevo conjunto que contiene solo los elementos que están en el conjunto original, pero no en el conjunto proporcionado como argumento.

```
val set1 = setOf("apple", "banana", "orange")
val set2 = setOf("banana", "pear")

val subtractSet = set1.subtract(set2)
println(subtractSet) // Output: ["apple", "orange"]
```

4. **containsAll()**: esta función verifica si todos los elementos del conjunto proporcionado como argumento se encuentran en el conjunto original.

```
val set1 = setOf("apple", "banana", "orange")
val set2 = setOf("banana", "pear")

val containsAll = set1.containsAll(set2)
println(containsAll) // Output: false
```

5. max() y min(): estas funciones devuelven el elemento máximo y mínimo del conjunto, respectivamente.

```
val set = setOf(3, 6, 2, 9, 4)

val maxElement = set.max()
val minElement = set.min()

println(maxElement) // Output: 9
println(minElement) // Output: 2
```

6. **toMutableSet()**: esta función convierte un conjunto inmutable en un conjunto mutable.

```
val immutableSet = setOf("apple", "banana", "orange")
val mutableSet = immutableSet.toMutableSet()

mutableSet.add("pear")
println(mutableSet) // Output: ["apple", "banana", "orange", "pear"]
```

▼ 5. Mapas en Kotlin

▼ ¿Qué es un mapa?

Son estructura de datos que permite asociar pares clave-valor, donde cada clave es única en el mapa y se utiliza para acceder al valor correspondiente. Se pueden crear mapas inmutables con la función mapOf() y mapas mutables con mutableMapOf(). Los valores en un mapa pueden ser de cualquier tipo, siempre y cuando la clave sea única.

▼ Creación de mapas en Kotlin.

Para crear mapas utilizamos la función mapOf O mutableMapOf.

La función mapOf() se utiliza para crear mapas inmutables. Toma una lista de pares clave-valor y devuelve un mapa inmutable. Por ejemplo:

```
val map = mapOf("key1" to "value1", "key2" to "value2", "key3" to "value3")
```

La función mutableMapOf() se utiliza para crear mapas mutables. Toma una lista de pares clave-valor y devuelve un mapa mutable. Por ejemplo:

```
val mutableMap = mutableMapOf("key1" to "value1", "key2" to "value2", "key3" to "value3")
```

▼ Accediendo a los elementos de un mapa

Para acceder a un mapa utilizamos la clave asociada a cada valor se utilizan la notacion de corchetes para los map inmutables y para los mutales se utiliza los corchetes, funcion get() y funcion put().

```
// map inmutables
val map = mapOf("key1" to "value1", "key2" to "value2")
println(map["key1"]) // Output: "value1"
println(map["key2"]) // Output: "value2"

// map inmutables
val mutableMap = mutableMapOf("key1" to "value1", "key2" to "value2")
println(mutableMap["key1"]) // Output: "value1"
println(mutableMap.get("key2")) // Output: "value2"
println(mutableMap.put("key3", "value3")) // Output: "value3"
```

▼ Modificando los elementos de un mapa

1. Para modificar un valor existente, se puede acceder al elemento a través de su clave y asignarle un nuevo valor:

```
val map = mutableMapOf("a" to 1, "b" to 2, "c" to 3)
map["b"] = 4 // modifica el valor asociado a la clave "b" a 4
```

2. Para agregar un nuevo elemento al mapa, se puede asignar un nuevo valor a una clave que no existía previamente:

```
val map = mutableMapOf("a" to 1, "b" to 2, "c" to 3)
map["d"] = 4 // agrega un nuevo elemento con la clave "d" y el valor 4
```

3.Para eliminar un elemento del mapa, se puede utilizar el método **remove** y especificar la clave del elemento que se desea eliminar:

```
val map = mutableMapOf("a" to 1, "b" to 2, "c" to 3)
map.remove("b") // elimina el elemento con la clave "b"
```

▼ Recorriendo un mapa

1.Recorrido por pares clave-valor utilizando un bucle for:

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
for ((key, value) in map) {
    println("La clave es $key y el valor es $value")
}
```

2.Recorrido utilizando la función forEach:

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
map.forEach { (key, value) ->
    println("La clave es $key y el valor es $value")
}
```

3. Recorrido utilizando la función **forEachIndexed**:

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
map.forEachIndexed { index, (key, value) ->
    println("El elemento $index tiene la clave $key y el valor $value")
}
```

▼ Funciones útiles para trabajar con mapas en Kotlin.

1. mapOf: función que crea un mapa inmutable a partir de pares clave-valor.

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

2.mutableMapOf: función que crea un mapa mutable a partir de pares clave-valor.

```
val map = mutableMapOf("a" to 1, "b" to 2, "c" to 3)
```

3.toMap: función de extensión que convierte una colección de pares clave-valor en un mapa.

```
val list = listOf("a" to 1, "b" to 2, "c" to 3)
val map = list.toMap()
```

4.keys: propiedad que devuelve una colección de las claves del mapa.

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
val keys = map.keys // devuelve ["a", "b", "c"]
```

5.values: propiedad que devuelve una colección de los valores del mapa.

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
val values = map.values // devuelve [1, 2, 3]
```

6. c**ontainsKey**: función que devuelve true si el mapa contiene la clave especificada.

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
val containsKey = map.containsKey("a") // devuelve true
```

7. **containsValue**: función que devuelve true si el mapa contiene el valor especificado.

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
val containsValue = map.containsValue(2) // devuelve true
```

8.**getOrDefault**: función que devuelve el valor asociado a la clave especificada, o un valor por defecto si la clave no existe en el mapa.

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
val value = map.getOrDefault("d", 0) // devuelve 0
```

9.**getOrElse:** función que devuelve el valor asociado a la clave especificada, o un valor calculado por una función lambda si la clave no existe en el mapa.

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
val value = map.getOrElse("d") { 0 } // devuelve 0
```

10.filter : función que devuelve un nuevo mapa que contiene solo los elementos que cumplen con una condición especificada en una función lambda.

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
val filteredMap = map.filter { (key, value) -> key != "b" && value > 1 }
// devuelve {"c" to 3}
```

▼ 6. Pares en Kotlin

▼ ¿Qué es un par?

un par es un objeto que contiene dos elementos y se utiliza para empaquetar dos valores relacionados. Se puede crear un par utilizando la función to, y se puede acceder a sus elementos utilizando las propiedades first y second. Los pares son útiles cuando se necesita trabajar con dos valores relacionados juntos en un contexto donde no tiene sentido definir una nueva clase para encapsular estos valores.

▼ Creación de pares en Kotlin

Se pueden crear pares utilizando la función to , que toma dos valores y los convierte en un objeto Pair.

```
// se puede crear un par que contenga una cadena de texto y un número entero.
val pair = "clave" to 10
// tambien se puede crear utilizando la funcion Pair que requiere de los dos
// valor para crear el objeto pair
val pair = Pair("clave", 10)
```

▼ Accediendo a los elementos de un par

```
val miPar = Pair(42, "Hola mundo")
// first accede al primer valor de Pair
val primerElemento = miPar.first // devuelve 42
// second accede al segundo valor de Pair
val segundoElemento = miPar.second // devuelve "Hola mundo"
// También se pueden asignar los elementos del par a variables individuales
// usando la sintaxis de desestructuración:
val (primerElemento, segundoElemento) = miPar
```

▼ Modificando los elementos de un par

Los pares son estructuras de datos inmutables, lo que significa que no se puede modificar.

Sin embargo si deseas modificar 1 o ambos datos utilizas la funcion copy() lo que hace es crear un nuevo par con los datos originales o actulizados de un par.

```
val miPar = Pair(42, "Hola mundo")
val miNuevoPar = miPar.copy(second = "Hola Kotlin")
```

▼ Recorriendo un par

Se puede recorrer un par utilizando la función forEach y una función lambda con dos parámetros para acceder a los elementos del par.

```
val miPar = Pair(42, "Hola mundo")
miPar.forEach { (primerElemento, segundoElemento) ->
    println("El primer elemento es $primerElemento y el segundo elemento es $segundoElemento")
}
```

de otra forma se puede acceder a los elementos de un par utilizando el first(que accede al primer valor) y second(que accede al segundo valor).

```
val miPar = Pair(42, "Hola mundo")
println("El primer elemento es ${miPar.first}")
println("el segundo elemento es ${miPar.second}")
```