# Estructuras de Datos en Kotlin

***Informe elaborado por la Aprendiz Valeria Sanchez Restrepo de la ficha 2469285 del programa ADSI.***

## 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 en Kotlin son herramientas eficientes y esenciales en la programación, ya que nos ayuda a organizar y/o acceder a los datos de manera accesible y efectiva. Estas se utilizan para una amplia variedad de aplicaciones de software, que mejoran la eficiencia, el rendimiento y la escalabilidad de los programas.

▼ **b.** Ventajas de utilizar estructuras de datos en Kotlin

> Algunas ventajas de utilizar estructuras de datos en Kotlin son:
>
> - Mejora la eficiencia, ya que estas estructuras de datos optimizan una forma en la que se almacena y se pueden acceder a los datos, es decir lo que hace que mejore el rendimiento y la eficiencia del programa.
> - Facilita mucho mas el código, ya que el lenguaje de Kotlin es conciso, fácil de leer y/o de interpretar e implementar, lo que hace que reduzca la cantidad de código necesario.
> - Ofrece una gran amplia variedad de opciones, ya que implementa estructuras de datos básicas y avanzadas que flexibilizan y adaptan las necesidades de cada proyecto.
> - Mejora la escalabilidad de las aplicaciones cuya función es que las estructuras de datos se adapten a medida de que los requisitos del proyecto se cambien.

▼ **c.** Diferencias entre las estructuras de datos en Kotlin y Java

> Sabemos que Kotlin y Java son lenguajes de programación con muchas similitudes al igual que estas comparten las estructuras de datos, pero sus diferencias son:

- En Kotlin las referencias nulas tienen funciones integradas para que estas sean mas seguras y así se eviten los errores en la programación. A diferencia de Java, estas referencias nulas deben ser verificadas para así evitar errores.
- En las sintaxis simplificadas, Kotlin utiliza sintaxis mas concisas y fáciles de leer, a diferencia de Java. Ya que Kotlin lo que hace es facilitar el leer y escribir el código que se utilizan en la estructuras de datos.
- En Kotlin admite la `Extencion functions`, estas son funciones adicionales que se agregan a las estructuras de datos ya existentes que hace que el código sea fácil de leer y mantenerlo, en cambio Java no admite esta extencion.
- Y por ultimo están las Estructuras de datos inmutables, que en Kotlin es un soporte nativo, que, una vez que se elabora una estructura datos, esta no se puede modificar. Lo que hace que el código sea mas seguro y evita esos errores en la programación.

## 2. Arreglos en Kotlin

▼ **a.** ¿Qué es un arreglo?

> Es una estructura de datos que se utiliza para almacenar elementos de una única variable, como una lista de números o una lista de palabras.

▼ **b.** Creación de arreglos en Kotlin

> En Kotlin se pueden crear arreglos de varias maneras como:

1. Arreglo vacío:

```
val array = arrayOfNulls<Int>(5) // crea un arreglo vacío de Ints con 5 elementos
```

2. Arreglo con elementos inicializados:

```
val array = arrayOf(1, 2, 3, 4, 5) // crea un arreglo de Ints con elementos inicializados
```

3. arreglo utilizando una expresión lambda:

```
val array = Array(5) { i -> i * 2 } // crea un arreglo de Ints con 5 elementos que son el doble de su índice
```

4. Arreglo de un tipo específico utilizando la función factory:

```
val array = IntArray(5) // crea un arreglo de Ints con 5 elementos
```

5. arreglo de varias dimensiones:

```
val array = Array(2) { Array(3) { i, j -> i * j } } // crea un arreglo de 2x3 de Ints que son el producto de sus índices
```

---

> Ya que se haya creado un arreglo, en el, se pueden acceder a los elementos utilizando el indice en el arreglo, como:

```
val array = arrayOf(1, 2, 3, 4, 5)
val firstElement = array[0] // obtiene el primer elemento del arreglo (1)
```

---

> En resumen, estas son las formas(arreglos vacíos, con elementos, con expresión lambda, función factory y dimensiones)en las que se pueden hacer arreglos en el lenguaje de Kotlin, y una vez creados estos arreglos se pueden acceder a estos elementos utilizando el indice del arreglo.

▼ **c.** Accediendo a los elementos de un arreglo

> Para acceder a los elementos de un arreglo se debe utilizar su indice en el arreglo. Al utilizar el indice este mismo comienza en 0, lo que hace es que el primer elemento del arreglo indica que su indice principal es el cero, y el segundo elemento tiene como indice el 1, y asi sucesivamente con cada elemento.

> Para acceder al elemento especifico del arreglo, en esta se pueden utilizar la notación de corchetes para poder pasar el indice del elemento entre ellos por ejemplo:

- Si se tiene el siguiente arreglo:

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

- Se puede acceder al primer elemento del arreglo, que es el 10, usando el siguiente código:

```
val primerNumero = numeros[0] // primerNumero tendrá el valor de 10
```

- También se puede modificar el elemento especifico del arreglo utilizando el mismo enfoque. Por ejemplo, para cambiar el segundo elemento del arreglo a 25, se puede usar el siguiente código:

```
numeros[1] = 25 // el segundo elemento del arreglo ahora es 25
```

- También es posible acceder a los elementos de un arreglo utilizando un bucle. Por ejemplo, para imprimir todos los elementos del arreglo en la consola, se puede usar un bucle for:

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

- Esto imprimirá los elementos del arreglo en orden:

```
10
25
30
40
50
```

> En resumen, en Kotlin para poder acceder a los elementos de un arreglo se utiliza la notación de corchetes y pasando al indice del elemento que se desea acceder o modificar. Al igual que para poder acceder a los elementos del arreglo se utiliza el bucle for.

▼ **d.** Modificando los elementos de un arreglo

> Para modificar los elementos de un arreglo utilizando la notación que son corchetes y pasando el índice del elemento que se desea modificar.

- Ejemplo del siguiente arreglo:

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

- Se puede modificar el segundo elemento del arreglo, que es el numero 20, para que sea el numero 25, utilizando el siguiente código:

```
numeros[1] = 25 // El segundo elemento del arreglo ahora es 25
```

- También se puede modificar los elementos múltiples del arreglo utilizando un bucle for. Ejemplo, para agregar 5 a cada elemente del arreglo, se puede usar el siguiente código:

```
for (i in numeros.indices) {
    numeros[i] += 5
}
```

- Esto lo que agregara son 5 a cada elemento del arreglo, por lo que el arreglo resultante será:

```
[15, 30, 35, 45, 55]
```

▼ **e.** Recorriendo un arreglo

> En Kotlin, se pueden recorrer un arreglo utilizando un bucle for, ya sea para acceder a cada uno de los elementos de un arreglo o para realizar alguna operación sobre ellos.

- Existen dos formas muy comunes para recorrer un arreglo en Kotlin:

1. Recorrer el arreglo por su indice: se utiliza un bucle for que itera a través de los indices del arreglo. Por cada iteracion, se puede acceder al elemento del arreglo en el indice actual utilizando la notación de corchetes.

- Por ejemplo, si tenemos el siguiente arreglo:

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

- Se podrá recorrer el arreglo utilizando su indice de la siguiente manera:

```
for (i in numeros.indices) {
    println("El elemento en el índice $i es ${numeros[i]}")
}
```

- Este código imprimirá lo siguiente en la consola:

```
El elemento en el índice 0 es 10
El elemento en el índice 1 es 20
El elemento en el índice 2 es 30
El elemento en el índice 3 es 40
El elemento en el índice 4 es 50
```

2. Recorrer el arreglo por sus elementos: se utiliza el bucle for que itera a través de cada elemento de un arreglo. Por cada iteracion, se puede acceder al elemento actual del arreglo utilizando una variable temporal.

- Por ejemplo, si tenemos el mismo arreglo:

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

- Podemos recorrer el arreglo utilizando sus elementos de la siguiente manera:

```
for (numero in numeros) {
    println("El número actual es $numero")
}
```

- Este código imprimirá lo siguiente en la consola:

```
El número actual es 10
El número actual es 20
El número actual es 30
El número actual es 40
El número actual es 50
```

> En resumen, se pueden recorrer los arreglos en Kotlin utilizando un bucle for. Se puede recorrer el arreglo por su indice o por sus elementos, dependiendo de la necesidad de la aplicación.

▼ **f.** Funciones útiles para trabajar con arreglos en Kotlin

> En Kotlin, existen varias funciones útiles que pueden ser utilizadas para trabajar con arreglos, Por ejemplo algunas funciones mas comunes:

1. `arrayOf()`: Esta función es utilizada para crear un nuevo arreglo. Se puede pasar una lista de elementos como argumentos de la función, y la función devolverá un arreglo que contiene los elementos pasados como argumentos.

- Por ejemplo:

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

2. `size` : Esta función devuelve el tamaño del arreglo. Por ejemplo:

```
val numeros = arrayOf(10, 20, 30, 40, 50)
val tamano = numeros.size // tamano = 5
```

3. `get(index: Int)` : Esta función se utiliza para acceder al elemento del arreglo en un indice especifico. Por ejemplo:

```
val numeros = arrayOf(10, 20, 30, 40, 50)
val elemento = numeros.get(2) // elemento = 30
```

4. `set(index: Int, value: T)` : Esta función se utiliza para modificar el valor de un elemento en un indice especifico del arreglo. Por ejemplo:

```
val numeros = arrayOf(10, 20, 30, 40, 50)
numeros.set(2, 35) // El tercer elemento del arreglo es ahora 35
```

5. `max()` : Esta función devuelve el valor máximo del arreglo. La función debe ser llamada en un arreglo de elementos comparables (por ejemplo, números o caracteres). Por ejemplo:

```
val numeros = arrayOf(10, 20, 30, 40, 50)
val maximo = numeros.max() // maximo = 50
```

6. `min()` : Esta función devuelve el valor mínimo del arreglo. La función debe ser llamada en un arreglo de elementos comparables (por ejemplo, números o caracteres). Por ejemplo:

```
val numeros = arrayOf(10, 20, 30, 40, 50)
val minimo = numeros.min() // minimo = 10
```

7. `sorted()` : Esta función devuelve un nuevo arreglo que contiene los elementos del arreglo original ordenados en orden ascendente. Por ejemplo:

```
val numeros = arrayOf(50, 20, 40, 10, 30)
val numerosOrdenados = numeros.sorted() // numerosOrdenados = [10, 20, 30, 40, 50]
```

8. `filter()` : Esta función devuelve un nuevo arreglo que contiene sólo los elementos del arreglo original que cumplen una determinada condición. La condición se especifica utilizando una función lambda. Por ejemplo:

```
val numeros = arrayOf(10, 20, 30, 40, 50)
val numerosPares = numeros.filter { it % 2 == 0 } // numerosPares = [10, 20, 40]
```

> En resumen, Kotlin proporciona una serie de funciones útiles para trabajar con arreglos, como la creación de un nuevo arreglo, el obtener el tamaño del arreglo, la obtención y modificación de elementos, la obtención de valores máximo y mínimo, la ordenación y la filtración de elementos.

## 3. Listas en Kotlin

▼ **a.** ¿Qué es una lista?

> En Kotlin una lista es una colección de elementos ordenados y accesibles mediante un indice. Al igual que los arreglos, las listas son estructuras de datos que permiten almacenar un conjunto de elementos del mismo tipo.

▼ **b.** Creación de listas en Kotlin

> En Kotlin, las listas se pueden crear utilizando la clase `List` o alguna de sus subclases, como `MutableList` . La clase `List` representa una lista de sólo lectura, lo que significa que una vez que se ha creado una lista, no se pueden agregar, eliminar o modificar elementos. Por otro lado, la clase `MutableList` representa una lista que se puede modificar.

- Aquí hay un ejemplo de cómo crear una lista de sólo lectura en Kotlin:

```
val lista = listOf("manzana", "banana", "naranja", "piña")
```

> En este ejemplo, se crea una lista de frutas utilizando la función `listOf()` . La lista es de sólo lectura, lo que significa que los elementos no se pueden agregar, eliminar o modificar.

- Por otro lado, aquí hay un ejemplo de cómo crear una lista mutable en Kotlin:

```
val listaMutable = mutableListOf("manzana", "banana", "naranja", "piña")
```

> En este ejemplo, se crea una lista mutable utilizando la función `mutableListOf()` .
>
> La lista es mutable, lo que significa que los elementos se pueden agregar, eliminar y modificar.

> En general, las listas son una estructura de datos muy útil en Kotlin, ya que permiten almacenar y manipular colecciones de elementos de manera dinámica.

▼ **c.** Accediendo a los elementos de una lista

> En Kotlin, se puede acceder a los elementos de una lista utilizando un índice, de manera similar a como se accede a los elementos de un arreglo. Para acceder a un elemento en particular, se utiliza la sintaxis `lista[indice]` , donde `lista` es el nombre de la lista y `indice` es el indice del elemento que se quiere acceder.

- Por ejemplo, supongamos que tenemos la siguiente lista de frutas:

```
val lista = listOf("manzana", "banana", "naranja", "piña")
```

> Para acceder al primer elemento de la lista (es decir, "manzana"), se utiliza la sintaxis `lista[0]` . Para acceder al segundo elemento (es decir, "banana"), se utiliza `lista[1]` , y así sucesivamente. Es importante tener en cuenta que si se intenta acceder a un elemento que está fuera de los límites de la lista, se producirá una excepción `IndexOutOfBoundsException` . Por ejemplo, si se intenta acceder al quinto elemento de la lista `lista[4]` ,
>
> se producirá una excepción, ya que la lista sólo tiene cuatro elementos.
>
> En resumen, para acceder a los elementos de una lista en Kotlin, se utiliza la sintaxis `lista[indice]` , donde `lista` es el nombre de la lista y `indice` es el índice del elemento que se quiere acceder.

▼ **d.** Modificando los elementos de una lista

> En Kotlin, se pueden modificar los elementos de una lista mutable utilizando la sintaxis `lista[indice] = valor` , donde `lista` es el nombre de la lista, `indice` es el índice del elemento que se quiere modificar y `valor` es el nuevo valor que se desea asignar al elemento.

- Por ejemplo, supongamos que tenemos la siguiente lista mutable de frutas:

```
val lista = mutableListOf("manzana", "banana", "naranja", "piña")
```

- Para modificar el segundo elemento de la lista (es decir, "banana") y cambiarlo por "melón", se puede usar la sintaxis `lista[1] = “melón”` :

```
lista[1] = "melón"
```

- Después de ejecutar esta línea de código, la lista quedará de la siguiente manera:

```
["manzana", "melón", "naranja", "piña"]
```

- También se puede utilizar la función `set()` para modificar un elemento en particular de una lista mutable. La sintaxis para utilizar esta función es la siguiente: `lista.set(indice, valor)` . Por ejemplo, para cambiar el tercer elemento de la lista por "kiwi", se puede utilizar la siguiente línea de código:

```
lista.set(2, "kiwi")
```

- Después de ejecutar esta línea de código, la lista quedará de la siguiente manera:

```
["manzana", "melón", "kiwi", "piña"]
```

> Es importante tener en cuenta que si se intenta modificar un elemento que está fuera de los límites de la lista, se producirá una excepción `IndexOutOfBoundsException` . También es importante tener en cuenta que sólo se pueden modificar los elementos de una lista mutable, no de una lista de sólo lectura.

▼ **e.** Recorriendo una lista

> En Kotlin, se pueden recorrer los elementos de una lista utilizando diferentes técnicas. A continuación se describen algunas de las más comunes:

1. **Bucle for:** se puede recorrer una lista utilizando un bucle for, de la siguiente manera:

```
val lista = listOf("manzana", "banana", "naranja", "piña")

for (elemento in lista) {
    println(elemento)
}
```

- Este código imprimirá cada uno de los elementos de la lista en la consola.

2. **Función forEach:** se puede utilizar la función `forEach` para recorrer los elementos de una lista y realizar alguna acción con cada uno de ellos. La sintaxis de la función es la siguiente: `lista.forEach { elemento → // hacer algo con el elemento }` . Por ejemplo, para imprimir cada uno de los elementos de la lista en la consola, se puede utilizar la siguiente línea de código:

```
lista.forEach { elemento ->
    println(elemento)
}
```

3. Índices de la lista: se puede utilizar la propiedad `indices` de la lista para recorrer los elementos de la lista junto con sus índices. La sintaxis es la siguiente:

```
for (indice in lista.indices) {
    println("El elemento en la posición $indice es ${lista[indice]}")
}
```

- Este código imprimirá cada uno de los elementos de la lista junto con su índice en la consola.

4. **Función map:** la función `map` se utiliza para aplicar una transformación a cada uno de los elementos de la lista y retornar una nueva lista con los elementos transformados. Por ejemplo, para crear una nueva lista con todos los elementos de la lista original en mayúsculas, se puede utilizar la siguiente línea de código:

```
val listaMayusculas = lista.map { elemento -> elemento.toUpperCase() }
```

- La nueva lista `listaMayusculas` tendrá los elementos "MANZANA", "BANANA", "NARANJA" y "PIÑA".

> Estas son sólo algunas de las técnicas que se pueden utilizar para recorrer una lista en Kotlin. La elección de la técnica dependerá del caso de uso particular y de las necesidades del programador.

▼ **f.** Funciones útiles para trabajar con listas en Kotlin

> Kotlin proporciona varias funciones útiles para trabajar con listas. A continuación se describen algunas de las más comunes:

1. **Función filter:** se utiliza para filtrar los elementos de una lista según una condición. La sintaxis de la función es la siguiente: `lista.filter { elemento → //condicion }` . Por ejemplo, para crear una nueva lista que contenga sólo las frutas que empiezan con la letra "m", se puede utilizar la siguiente línea de código:

```
val listaFiltrada = lista.filter { elemento -> elemento.startsWith("m") }
```

- La nueva lista `listaFiltrada` endrá sólo el elemento "manzana".

2. **Función find:** se utiliza para encontrar el primer elemento de una lista que cumpla con una condición. La sintaxis de la función es la siguiente: `lista.find{ elemento → //condicion }` . Por ejemplo, para encontrar el primer elemento de la lista que contenga la letra "a", se puede utilizar la siguiente línea de código:

```
val elementoEncontrado = lista.find { elemento -> elemento.contains("a") }
```

- La variable `elementoEncontrado` tendrá el valor "manzana", que es el primer elemento de la lista que contiene la letra "a".

3. **Función count:** se utiliza para contar los elementos de una lista que cumplan con una condición. La sintaxis de la función es la siguiente: `lista.count { elemento → //condicion }` . Por ejemplo, para contar el número de frutas en la lista que contienen la letra "n", se puede utilizar la siguiente línea de código:

```
val cantidadElementos = lista.count { elemento -> elemento.contains("n") }
```

- La variable `cantidadElementos` tendrá el valor 2, que es el número de frutas en la lista que contienen la letra "n".

4. **Función sorted:** se utiliza para ordenar los elementos de una lista. Por defecto, la función ordena los elementos en orden ascendente. La sintaxis de la función es la siguiente: `lista.sorted()` . Por ejemplo, para ordenar la lista en orden alfabético, se puede utilizar la siguiente línea de código:

```
val listaOrdenada = lista.sorted()
```

- La nueva lista `listaOrdenada` tendrá los elementos en el siguiente orden: "banana", "manzana", "naranja" y "piña".

5. **Función groupBy:** se utiliza para agrupar los elementos de una lista según una determinada condición. La sintaxis de la función es la siguiente: `lista.groupBy { elemento → //condicion }` . Por ejemplo, para agrupar las frutas de la lista según su longitud, se puede utilizar la siguiente línea de código:

```
val frutasPorLongitud = lista.groupBy { elemento -> elemento.length }
```

- La variable `frutasPorLongitud` será un mapa cuyas claves son las longitudes de las frutas, y cuyos valores son listas de frutas con la misma longitud.

> Estas son sólo algunas de las funciones útiles que se pueden utilizar para trabajar con listas en Kotlin. La elección de la función dependerá del caso de uso particular y de las necesidades del programador.

## 4. Conjuntos en Kotlin

▼ **a.** ¿Qué es un conjunto?

> En Kotlin, un conjunto (set en inglés) es una colección de elementos que no se pueden repetir y cuyo orden no está definido. Los conjuntos se utilizan cuando se necesita almacenar una colección de elementos únicos sin importar su orden.
>
> Los conjuntos se definen en Kotlin utilizando la interfaz Set, que puede tener implementaciones específicas, como HashSet o TreeSet. Un HashSet es una implementación que no mantiene un orden específico de los elementos, mientras que un TreeSet mantiene los elementos ordenados.

▼ **b.** Creación de conjuntos en Kotlin

> Para crear un conjunto en Kotlin, se puede utilizar la función setOf, que devuelve un conjunto inmutable de los elementos dados como argumento. Por ejemplo, el siguiente código crea un conjunto de frutas:

```
val frutas = setOf("manzana", "banana", "naranja", "piña")
```

- También se puede crear un conjunto mutable utilizando la función mutableSetOf. Por ejemplo:

```
val frutasMutable = mutableSetOf("manzana", "banana", "naranja", "piña")
```

- En un conjunto, los elementos se pueden agregar o eliminar utilizando las funciones add y remove. Por ejemplo:

```
frutasMutable.add("mango") // Agrega el elemento "mango" al conjunto
frutasMutable.remove("naranja") // Elimina el elemento "naranja" del conjunto
```

- Los conjuntos también proporcionan operaciones como union, intersection y subtract, que permiten realizar operaciones de conjuntos como la unión, la intersección y la diferencia. Por ejemplo:

```
val numeros1 = setOf(1, 2, 3)
val numeros2 = setOf(2, 3, 4)
val union = numeros1.union(numeros2) // Devuelve un conjunto que contiene los elementos de ambos conjuntos sin repetición
val interseccion = numeros1.intersect(numeros2) // Devuelve un conjunto que contiene los elementos que están en ambos conjuntos
val diferencia = numeros1.subtract(numeros2) // Devuelve un conjunto que contiene los elementos de numeros1 que no están en numeros2
```

> En resumen, un conjunto en Kotlin es una colección de elementos únicos sin orden definido. Se pueden agregar y eliminar elementos, y se pueden realizar operaciones de conjuntos para combinar, intersectar o restar conjuntos.

▼ **c.** Accediendo a los elementos de un conjunto

> En un conjunto de Kotlin, los elementos no tienen un índice asociado, ya que no están en un orden definido. Por lo tanto, para acceder a los elementos de un conjunto, se puede utilizar un ciclo for-each o la función forEach, que permite iterar sobre cada elemento del conjunto y ejecutar una acción para cada elemento.

- Por ejemplo, si tenemos el siguiente conjunto de números:

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

- Podemos recorrer el conjunto e imprimir cada número de la siguiente manera:

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

- Esto imprimirá:

```
1
2
3
4
5
```

- También es posible convertir un conjunto a una lista para poder acceder a sus elementos por índice. Por ejemplo:

```
val numeros = setOf(1, 2, 3, 4, 5)
val listaNumeros = numeros.toList()
println(listaNumeros[0]) // Imprime el primer elemento del conjunto (1)
```

> Sin embargo, hay que tener en cuenta que si se requiere acceder a los elementos de un conjunto por índice, probablemente no sea la estructura de datos más adecuada para la tarea, ya que la idea principal de un conjunto es tener una colección de elementos únicos sin un orden específico.

▼ **d.** Modificando los elementos de un conjunto

> En un conjunto de Kotlin, los elementos son inmutables, es decir, no se pueden modificar directamente después de haber sido agregados. Si se desea modificar un elemento en particular, es necesario eliminar el elemento y agregar uno nuevo en su lugar.

- Por ejemplo, si tenemos el siguiente conjunto de frutas:

```
var frutas = mutableSetOf("manzana", "banana", "naranja", "piña")
```

- Y queremos cambiar la naranja por una pera, podemos hacerlo de la siguiente manera:

```
frutas.remove("naranja")
frutas.add("pera")
```

- Después de ejecutar estas líneas de código, el conjunto de frutas quedaría de la siguiente manera:

```
["manzana", "banana", "pera", "piña"]
```

- También se pueden modificar varios elementos a la vez utilizando las funciones de operación de conjuntos. Por ejemplo, si tenemos los siguientes conjuntos:

```
val numeros1 = mutableSetOf(1, 2, 3, 4, 5)
val numeros2 = setOf(4, 5, 6, 7)
```

- Y queremos que el conjunto numeros1 contenga los elementos de ambos conjuntos, podemos utilizar la función union:

```
numeros1 = numeros1.union(numeros2) as MutableSet<Int>
```

- Después de ejecutar esta línea de código, el conjunto numeros1 quedaría de la siguiente manera:

```
[1, 2, 3, 4, 5, 6, 7]
```

> En resumen, en un conjunto de Kotlin los elementos son inmutables, por lo que no se pueden modificar directamente después de haber sido agregados. Si se necesita cambiar un elemento, es necesario eliminar el elemento y agregar uno nuevo en su lugar.

▼ **e.** Recorriendo un conjunto

> En un conjunto de Kotlin, los elementos no tienen un orden definido, por lo que no se puede acceder a ellos por índice como en una lista. En cambio, se puede recorrer un conjunto utilizando un ciclo for-each o la función forEach, que permite iterar sobre cada elemento del conjunto y ejecutar una acción para cada elemento.

- Por ejemplo, si tenemos el siguiente conjunto de frutas:

```
val frutas = setOf("manzana", "banana", "naranja", "piña")
```

- Podemos recorrer el conjunto e imprimir cada fruta de la siguiente manera:

```
frutas.forEach { fruta ->
    println(fruta)
}
```

- Esto imprimirá:

```
manzana
banana
naranja
piña
```

- También se puede utilizar un ciclo for-each para recorrer el conjunto:

```
for (fruta in frutas) {
    println(fruta)
}
```

Esto imprimirá el mismo resultado que el código anterior.

> Es importante tener en cuenta que en un conjunto, los elementos no tienen un orden definido y pueden aparecer en cualquier orden durante la iteración. Si se requiere que los elementos se iteren en un orden específico, se puede convertir el conjunto a una lista y luego iterar sobre la lista.

▼ **f.** Funciones útiles para trabajar con conjuntos en Kotlin

> En Kotlin, existen varias funciones útiles para trabajar con conjuntos. Algunas de las más comunes son:

- **add(elemento: E)**: Agrega el elemento especificado al conjunto.
- **remove(elemento: E)**: Elimina el elemento especificado del conjunto, si existe.
- **contains(elemento: E)**: Verifica si el conjunto contiene el elemento especificado.
- **isEmpty()**: Verifica si el conjunto está vacío.
- **size()**: Devuelve el número de elementos en el conjunto.
- **union(other: Set<E>)**: Devuelve un conjunto que contiene todos los elementos de este conjunto y del conjunto especificado.
- **intersect(other: Set<E>)**: Devuelve un conjunto que contiene los elementos que están presentes en este conjunto y en el conjunto especificado.
- **subtract(other: Set<E>)**: Devuelve un conjunto que contiene los elementos que están presentes en este conjunto pero no en el conjunto especificado.
- **filter(predicate: (E) -> Boolean)**: Devuelve un conjunto que contiene solo los elementos que cumplen con la condición especificada por el predicado.
- **map(transform: (E) -> R)**: Devuelve un conjunto que contiene los elementos transformados por la función especificada.

Por ejemplo, si tenemos el siguiente conjunto de frutas:

```
val frutas = mutableSetOf("manzana", "banana", "naranja", "piña")
```

- Podemos utilizar la función add para agregar una nueva fruta al conjunto:

```
frutas.add("kiwi")
```

- También podemos utilizar la función remove para eliminar una fruta del conjunto:

```
frutas.remove("naranja")
```

- Podemos utilizar la función contains para verificar si el conjunto contiene una fruta en particular:

```
if (frutas.contains("banana")) {
    println("El conjunto contiene banana")
}
```

- Podemos utilizar la función filter para crear un nuevo conjunto que contenga solo las frutas que empiezan con la letra "m":

```
val frutasConM = frutas.filter { it.startsWith("m") }
```

- También podemos utilizar la función map para crear un nuevo conjunto que contenga las frutas transformadas a mayúsculas:

```
val frutasEnMayuscula = frutas.map { it.toUpperCase() }
```

## 5. Mapas en Kotlin

▼ **a.** ¿Qué es un mapa?

> En Kotlin, un mapa es una colección de pares clave-valor donde cada clave está asociada a un valor. Es similar a una tabla de base de datos donde la clave sería la columna que se utiliza para buscar y el valor sería la información en esa fila correspondiente a la columna de la clave.
>
> Los mapas se utilizan para almacenar y recuperar información utilizando una clave en lugar de un índice. Cada clave en un mapa debe ser única y solo puede estar asociada con un valor. Los valores pueden ser duplicados.

▼ **b.** Creación de mapas en Kotlin

> En Kotlin, los mapas se pueden crear utilizando la interfaz Map, que es inmutable, o utilizando la interfaz MutableMap, que es mutable y permite agregar, actualizar y eliminar elementos.
>
> Los mapas se pueden inicializar utilizando la función mapOf() o mutableMapOf(), que aceptan una lista de pares clave-valor. Por ejemplo, el siguiente código crea un mapa de nombres de colores en inglés y sus códigos hexadecimales:

```
val colores = mapOf(
    "rojo" to "#FF0000",
    "verde" to "#00FF00",
    "azul" to "#0000FF"
)
```

En este caso, cada clave es un nombre de color en inglés y cada valor es el código hexadecimal correspondiente.

- Para acceder a un valor en un mapa, se utiliza la clave correspondiente. Por ejemplo, para obtener el código hexadecimal del color "rojo", se utiliza la clave "rojo" de la siguiente manera:

```
val codigoRojo = colores["rojo"]
```

En este caso, la variable `codigoRojo` contendrá el valor `#FF0000` .

> Además de acceder a valores, los mapas también permiten agregar nuevos pares clave-valor utilizando la función put(), actualizar valores existentes utilizando la misma función put() con una clave existente, y eliminar pares clave-valor utilizando la función remove().

▼ **c.** Accediendo a los elementos de un mapa

> Para acceder a los elementos de un mapa en Kotlin, se utiliza la clave correspondiente.

- Por ejemplo, si tenemos un mapa de colores y códigos hexadecimales, como el siguiente:

```
val colores = mapOf(
    "rojo" to "#FF0000",
    "verde" to "#00FF00",
    "azul" to "#0000FF"
)
```

- Podemos acceder al código hexadecimal del color "rojo" de la siguiente manera:

```
val codigoRojo = colores["rojo"]
```

En este caso, la variable `codigoRojo` contendrá el valor `#FF0000` .

- También es posible utilizar la función getValue() para acceder a un valor en un mapa y lanzar una excepción si la clave no existe en el mapa. Por ejemplo:

```
val codigoRojo = colores.getValue("rojo")
```

> En este caso, la variable `codigoRojo` contendrá el valor `#FF0000`. Si intentamos obtener el valor de una clave que no existe en el mapa utilizando la función getValue(), se lanzará una excepción `NoSuchElementException`.
>
> Por otro lado, si queremos acceder a un valor que podría no estar presente en el mapa, podemos utilizar la función get(), que devuelve el valor correspondiente a la clave si la clave existe, o un valor predeterminado si la clave no existe. Por ejemplo:

```
val codigoNegro = colores.get("negro", "#000000")
```

> En este caso, la variable `codigoNegro` contendrá el valor predeterminado `#000000`, ya que la clave "negro" no está presente en el mapa.

> Además, también es posible utilizar el operador `in` para verificar si una clave está presente en un mapa:

```
if ("rojo" in colores) {
    println("El código hexadecimal del rojo es ${colores["rojo"]}")
}
```

En este caso, si la clave "rojo" está presente en el mapa, se imprimirá el código hexadecimal correspondiente.

▼ **d.** Modificando los elementos de un mapa

> Para modificar los elementos de un mapa en Kotlin, se puede acceder al valor correspondiente a través de la clave y asignarle un nuevo valor. Por ejemplo, si tenemos un mapa de colores y códigos hexadecimales, como el siguiente:

```
var colores = mutableMapOf(
    "rojo" to "#FF0000",
    "verde" to "#00FF00",
```

```
    "azul" to "#0000FF"
)
```

- Podemos cambiar el código hexadecimal del color "verde" de la siguiente manera:

```
colores["verde"] = "#008000"
```

> En este caso, el valor del mapa para la clave "verde" será modificado a `#008000`.

- También se puede utilizar la función put() para agregar o modificar un valor en un mapa mutable. Por ejemplo:

```
colores.put("amarillo", "#FFFF00")
```

> En este caso, se agrega una nueva clave-valor al mapa, ya que "amarillo" no estaba presente anteriormente. Si la clave ya existe en el mapa, se sobrescribe el valor existente.

- Si queremos eliminar una clave y su valor correspondiente de un mapa mutable, podemos utilizar la función remove(). Por ejemplo:

```
colores.remove("azul")
```

- En este caso, la clave "azul" y su valor correspondiente serán eliminados del mapa.

▼ **e.** Recorriendo un mapa

> Para recorrer un mapa en Kotlin, podemos utilizar un bucle for que itere sobre los pares clave-valor del mapa.

- Si queremos recorrer un mapa inmutable, podemos utilizar el método forEach() del mapa. Por ejemplo, si tenemos el siguiente mapa:

```
val edades = mapOf(
    "Juan" to 25,
    "María" to 30,
    "Pedro" to 40
)
```

- Podemos imprimir cada par clave-valor de la siguiente manera:

```
edades.forEach { (nombre, edad) ->
    println("$nombre tiene $edad años")
}
```

- Este código imprimirá lo siguiente:

```
Juan tiene 25 años
```

- Si queremos recorrer un mapa mutable, podemos utilizar un bucle for que itere sobre las claves del mapa y acceda al valor correspondiente. Por ejemplo, si tenemos el siguiente mapa mutable:

```
val capitales = mutableMapOf(
    "España" to "Madrid",
    "Francia" to "París",
    "Italia" to "Roma"
)
```

- Podemos imprimir cada par clave-valor de la siguiente manera:

```
for ((pais, capital) in capitales) {
    println("La capital de $pais es $capital")
}
```

- Este código imprimirá lo siguiente:

```
La capital de España es Madrid
```

▼ **f.** Funciones útiles para trabajar con mapas en Kotlin

> En Kotlin, existen varias funciones útiles para trabajar con mapas, tanto inmutables como mutables. Algunas de ellas son:

1. **getOrDefault():** Esta función nos permite obtener el valor correspondiente a una clave, y si la clave no existe en el mapa, nos devuelve un valor por defecto que especifiquemos. Por ejemplo:

```
val edades = mapOf(
    "Juan" to 25,
    "María" to 30,
    "Pedro" to 40
)
val edadMaria = edades.getOrDefault("María", 0)
val edadLuis = edades.getOrDefault("Luis", 0)
```

- En este caso, la variable edadMaria tendrá el valor 30, ya que la clave "María" sí existe en el mapa. Por otro lado, la variable edadLuis tendrá el valor 0, ya que la clave "Luis" no existe en el mapa.

2. **keys():** Esta función nos devuelve un conjunto de las claves del mapa. Por ejemplo:

```
val edades = mapOf(
    "Juan" to 25,
    "María" to 30,
    "Pedro" to 40
)
val nombres = edades.keys
```

- En este caso, la variable nombres tendrá el valor `setOf("Juan", "María", "Pedro")`.

3. **values():** Esta función nos devuelve una lista de los valores del mapa. Por ejemplo:

```
val edades = mapOf(
    "Juan" to 25,
    "María" to 30,
    "Pedro" to 40
)
val listaEdades = edades.values
```

- En este caso, la variable listaEdades tendrá el valor `[25, 30, 40]`.

4. **filter():** Esta función nos permite filtrar los pares clave-valor del mapa según un predicado que especifiquemos. Por ejemplo:

```
val edades = mapOf(
    "Juan" to 25,
    "María" to 30,
    "Pedro" to 40
)
val mayoresDe30 = edades.filter { it.value > 30 }
```

- En este caso, la variable mayoresDe30 tendrá el valor `mapOf("Pedro" to 40)`, ya que este es el único par clave-valor del mapa cuyo valor es mayor a 30.

5. **toMutableMap():** Esta función nos permite convertir un mapa inmutable en un mapa mutable. Por ejemplo:

```
val edades = mapOf(
    "Juan" to 25,
    "María" to 30,
    "Pedro" to 40
)
val edadesMutable = edades.toMutableMap()
edadesMutable["Juan"] = 26
```

- En este caso, la variable edadesMutable será un mapa mutable con los mismos pares clave-valor que el mapa inmutable original. Luego, se modifica el valor correspondiente a la clave "Juan" a 26.

## 6. Pares en Kotlin

▼ a. ¿Qué es un par?

> En Kotlin, un par es una estructura de datos que permite almacenar dos valores relacionados entre sí. Se representa por el tipo de datos `Pair` y contiene dos componentes que se pueden acceder con las propiedades `first` y `second`.

> El primer componente de un par representa el valor que se relaciona con el primer elemento, mientras que el segundo componente representa el valor que se relaciona con el segundo elemento. Los componentes pueden ser de cualquier tipo de datos en Kotlin.

> Los pares se utilizan a menudo para devolver múltiples valores de una función, para pasar dos valores relacionados entre sí como un solo argumento, o para asociar dos valores en una estructura de datos simple.

▼ b. Creación de pares en Kotlin

- En Kotlin, un par se crea utilizando la función `Pair()`. La sintaxis para crear un par es la siguiente:

```
val miPar: Pair<TipoDeDato1, TipoDeDato2> = Pair(valor1, valor2)
```

> Donde `TipoDeDato1` y `TipoDeDato2` son los tipos de datos de los valores que se van a almacenar en el par, y `valor1` y `valor2` son los valores que se van a almacenar en el primer y segundo componente del par, respectivamente.

- Por ejemplo, para crear un par que almacena un nombre y una edad, se puede escribir:

```
val nombreEdad = Pair("Juan", 25)
```

- También se pueden utilizar las funciones de extensión `to` y `toPair` para crear un par de una manera más concisa. Por ejemplo:

```
val nombreEdad = "Juan" to 25 // utilizando la función to()
val otroPar = 3.14 to "pi" // también se pueden utilizar valores de diferentes tipos de datos
```

```
val listaPares = listOf("Juan" to 25, "Pedro" to 30, "María" to 27) // lista de pares
```

▼ c. Accediendo a los elementos de un par

- En Kotlin, se puede acceder a los elementos de un par utilizando las propiedades `first` y `second`.

```
val miPar = Pair("Juan", 25)

val nombre = miPar.first
val edad = miPar.second

println("El nombre es $nombre y la edad es $edad") // salida: "El nombre es Juan y la edad es 25"
```

- También se puede desestructurar un par en dos variables separadas utilizando la sintaxis de declaración de variables con el operador `in`:

```
val miPar = Pair("Juan", 25)

val (nombre, edad) = miPar

println("El nombre es $nombre y la edad es $edad") // salida: "El nombre es Juan y la edad es 25"
```

- En este ejemplo, la declaración de variables `val (nombre, edad) = miPar` crea dos variables `nombre` y `edad`, que se inicializan con los valores del primer y segundo componente del par, respectivamente.

▼ d. Modificando los elementos de un par

- En Kotlin, un par es inmutable, lo que significa que no se puede modificar una vez que se ha creado. Si se desea cambiar uno de los valores del par, es necesario crear un nuevo par con los valores actualizados.

```
var miPar = Pair("Juan", 25)

miPar = miPar.copy(second = 30)

val nombre = miPar.first
val edad = miPar.second

println("El nombre es $nombre y la edad es $edad") // salida: "El nombre es Juan y la edad es 30"
```

> En este ejemplo, se crea un par inmutable `miPar` con los valores `"Juan"` y `25`. Luego, se actualiza la edad del par a `30` utilizando la función `copy()` que devuelve un nuevo par con los mismos valores, excepto por el valor que se especifica en la llamada a `copy()`. Finalmente, se imprimen los valores del par actualizado utilizando las propiedades `first` y `second`.

▼ e. Recorriendo un par

- Como un par en Kotlin solo tiene dos elementos, no es necesario recorrerlo. Se puede acceder a los valores de sus elementos directamente a través de las propiedades `first` y `second`.
- Si se desea procesar un conjunto de pares, se puede utilizar un bucle `for` para recorrer una lista de pares.

```
val listaDePares = listOf(Pair("Juan", 25), Pair("Ana", 30), Pair("Luis", 40))

for ((nombre, edad) in listaDePares) {
    println("El nombre es $nombre y la edad es $edad")
}
```

- En este ejemplo, se crea una lista de pares `listaDePares` que contiene tres pares. Luego, se utiliza un bucle `for` para recorrer la lista, y en cada iteración del bucle, se desestructura el par en dos variables `nombre` y `edad`, que se imprimen en la consola.

▼ f. Funciones útiles para trabajar con pares en Kotlin

> En Kotlin, los pares son una estructura de datos simple que se utiliza para almacenar dos elementos relacionados. A continuación, se describen algunas funciones útiles para trabajar con pares en Kotlin:

- `Pair(first, second)`: función que crea un par con los dos valores especificados.
- `first` y `second`: propiedades que permiten acceder a los valores del par.
- `copy(first = nuevoValor1, second = nuevoValor2)`: función que crea un nuevo par con los mismos valores que el original, excepto por los valores especificados. Esta función devuelve un nuevo par inmutable.
- `toList()`: función que convierte un par en una lista de dos elementos.
- `toTriple()`: función que convierte un par en una tripleta, agregando un tercer valor `null`.

A continuación, se muestra un ejemplo que utiliza algunas de estas funciones:

```
val miPar = Pair("Juan", 25)

println("El nombre es ${miPar.first} y la edad es ${miPar.second}")

val nuevoPar = miPar.copy(second = 30)

println("El nuevo par es $nuevoPar")

val listaDePares = listOf(miPar, Pair("Ana", 30), Pair("Luis", 40))

val listaDeEdades = listaDePares.map { it.second }

println("Las edades son $listaDeEdades")

val tripleta = miPar.toTriple()

println("La tripleta es $tripleta")
```

- En este ejemplo, se crea un par inmutable `miPar` con los valores `"Juan"` y `25`. Se accede a los valores del par utilizando las propiedades `first` y `second`, y se imprime el resultado. Luego, se crea un nuevo par con el valor de edad actualizado utilizando la función `copy()`, y se imprime el resultado.
- A continuación, se crea una lista de pares `listaDePares`, y se utiliza la función `map()` para crear una lista de edades a partir de la lista de pares. Finalmente, se convierte el par `miPar` en una tripleta utilizando la función `toTriple()`, y se imprime el resultado.