Taller Estructuras de Datos en Kotlin

Introducción a las estructuras de datos en Kotlin

¿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 y efectiva, permitiendo el acceso y la manipulación de los datos de manera más rápida y sencilla. Se utilizan en programación para representar y gestionar datos complejos en una variedad de aplicaciones, como bases de datos, algoritmos de búsqueda y clasificación, procesamiento de imágenes, entre otros. Algunos ejemplos de estructuras de datos comunes incluyen arreglos, listas enlazadas, árboles y grafos.

Ventajas de utilizar estructuras de datos en Kotlin

Algunas ventajas de utilizar estructuras de datos en Kotlin son:

1. Mayor eficiencia y rendimiento: Las estructuras de datos bien diseñadas pueden mejorar la eficiencia y el rendimiento de la aplicación. Por ejemplo, el uso de un HashSet puede mejorar el rendimiento en búsquedas y eliminaciones de elementos.
2. Simplifica el código: Al usar estructuras de datos, se puede simplificar el código y reducir la cantidad de líneas necesarias para lograr ciertas tareas.
3. Mejora la legibilidad del código: Las estructuras de datos bien nombradas y organizadas pueden hacer que el código sea más fácil de leer y entender.
4. Reutilización de código: Almacenar datos en estructuras de datos hace que sea más fácil reutilizar código en diferentes partes de una aplicación.
5. Facilita el mantenimiento del código: Al utilizar estructuras de datos, el mantenimiento del código es más fácil ya que las operaciones con datos complejos están encapsuladas en la estructura de datos.

En resumen, las estructuras de datos son herramientas útiles que pueden mejorar la eficiencia, el rendimiento y la calidad del código en Kotlin.

Diferencias entre las estructuras de datos en Kotlin y C#

Kotlin y C# son lenguajes de programación diferentes, por lo que hay algunas diferencias en las estructuras de datos disponibles en cada uno. Aquí hay algunas diferencias:

1. Tipos de estructuras de datos: Kotlin y C# tienen algunas estructuras de datos en común, como Listas, Mapas y Conjuntos, pero también tienen algunas estructuras de datos únicas. Por ejemplo, Kotlin tiene la estructura de datos Rango (Range), mientras que C# tiene la estructura de datos Tuples.
2. Implementación: Las estructuras de datos en Kotlin y C# pueden implementarse de diferentes maneras. Por ejemplo, las Listas en Kotlin pueden ser mutables o inmutables, mientras que en C# las Listas son siempre mutables. Además, C# tiene estructuras de datos que son exclusivas de la plataforma .NET, como Stack y Queue.
3. Herencia y polimorfismo: Kotlin y C# manejan la herencia y el polimorfismo de manera diferente, lo que afecta a cómo se utilizan las estructuras de datos en cada lenguaje. En Kotlin, las Listas inmutables heredan de List, lo que permite tratar una Lista inmutable como una Lista. En C#, la herencia es menos común, y se utilizan más las interfaces para definir la funcionalidad de las estructuras de datos.

En resumen, aunque existen algunas similitudes entre las estructuras de datos en Kotlin y C#, también hay algunas diferencias en cuanto a los tipos de estructuras de datos, su implementación y la herencia y el polimorfismo.

Arreglos en Kotlin

▼ ¿Qué es un arreglo?

Es una estructura de datos que permite almacenar un conjunto de elementos del mismo tipo en una sola variable. Los elementos del arreglo se acceden mediante un índice numérico que indica su posición en el arreglo. Los arreglos son muy utilizados en programación para almacenar y manipular conjuntos de datos de manera eficiente.

▼ Creación de arreglos en Kotlin.

En Kotlin, se puede crear un arreglo utilizando la función **arrayOf()** o **array()** de la siguiente manera:

```
// Crear un arreglo de enteros
val numeros = arrayOf(1, 2, 3, 4, 5)

// Crear un arreglo de cadenas
val nombres = array("Juan", "María", "Pedro", "Lucía")
```

También es posible inicializar un arreglo vacío de una longitud especifica utilizando la función **Array()** de la siguiente manera:

```
// Crear un arreglo vacío de 5 elementos
val miArreglo = Array(5, {0})
```

> En este ejemplo, el primer parámetro es la longitud del arreglo y el segundo es una función lambda que define el valor inicial de cada elemento del arreglo. En este caso, todos los elementos se inicializan en cero.

▼ Accediendo a los elementos de un arreglo

Para acceder a los elementos de un arreglo, se usa el operador del índice **'[]'**. El índice del elemento que deseas acceder se especifica entre corchetes. y el primer elemento del arreglo tiene un índice de 0.

> Por ejemplo, en el ejemplo anterior tenemos un array llamado "**números**", para acceder al segundo elemento del arreglo ('2'), puedes hacerlo de la siguiente manera:

```
val segundoNumero = numeros[1]
```

El número entre corchetes ('1' en este caso) es el índice del segundo elemento del arreglo, ya que el primer elemento tiene un índice de 0.

▼ Modificando los elementos de un arreglo

Para modificar los elementos de un arreglo se usa también el operador del índice. Por ejemplo, para cambiar el valor del tercer elemento del arreglo ('3'), puedes hacer lo siguiente:

```
numeros[2] = 10
```

> Después de esta línea de código, el arreglo "**números**" se verá así: '**[1 ,2 ,10 ,4 ,5]**'

▼ Recorriendo un arreglo

Para recorrer un arreglo se puede usar el bucle `for` o la función de orden superior `forEach()`.

1. Con el bucle 'for': Puedes recorrer el arreglo utilizando un bucle `for` y accediendo a cada elemento del arreglo mediante su índice. Por ejemplo, el siguiente código recorre un arreglo de enteros llamado `números` e imprime cada

elemento en la consola:

```
val numeros = arrayOf(1, 2, 3, 4, 5)
for (i in numeros.indices) {
    println(numeros[i])
}
```

2. Con la función ‘forEach()’: Puedes recorrer el arreglo utilizando la función ‘forEach()’, que aplica una función a cada elemento del arreglo. La función ‘forEach()’ toma como argumento una función de orden superior que define la operación a realizar en cada elemento del arreglo. Por ejemplo, el siguiente código recorre un arreglo de cadenas llamado ‘nombres’ e imprime cada elemento en la consola:

```
val nombres = arrayOf("Juan", "Pedro", "María", "Lucía")
nombres.forEach { nombre ->
    println(nombre)
}
```

> En este ejemplo, la función de orden superior `{ nombre -> println(nombre) }` se aplica a cada elemento del arreglo `nombres` y simplemente imprime el nombre en la consola. Puedes definir cualquier función de orden superior para realizar una operación específica en cada elemento del arreglo.

▼ Funciones útiles para trabajar con arreglos en Kotlin

Existen varias funciones útiles para trabajar con arreglos. Algunas de las funciones más comunes son:

1. ‘size’: Devuelve el tamaño del arreglo

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

2. ‘get’: Devuelve el elemento del arreglo en la posición especificada.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val segundoNumero = numeros.get(1) // segundoNumero = 2
```

3. ‘set’: Cambia el valor del elemento del arreglo en la posición especificada.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
numeros.set(2, 10) // cambia el tercer elemento a 10
```

4. ‘indexOf’: Devuelve el índice de la primera ocurrencia del elemento especificado en el arreglo, o -1 si el elemento no se encuentra en el arreglo.

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

5. ´lastIndexOf’: Devuelve el índice de la última ocurrencia del elemento especificado en el arreglo, o -1 si el elemento no se encuentra en el arreglo.

```
val numeros = arrayOf(1, 2, 3, 4, 3)
val ultimoIndice = numeros.lastIndexOf(3) // ultimoIndice = 4
```

6. ‘contains’: Devuelve ‘true’ si el arreglo contiene el elemento especificado, o ‘false’ en caso contrario.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val contieneTres = numeros.contains(3) // contieneTres = true
```

7. ‘sum’: Devuelve la suma de todos los elementos del arreglo.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val suma = numeros.sum() // suma = 15
```

8. ‘average’: Devuelve el promedio de todos los elementos del arreglo.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val promedio = numeros.average() // promedio = 3.0
```

> Estas son solo algunas de las funciones útiles para trabajar con arreglos en Kotlin. Hay muchas más funciones disponibles, como `filter`, `map`, `reduce`, etc. que permiten realizar operaciones más complejas en los arreglos.

▼ Listas en Kotlin

▼ ¿Qué es una lista?

En Kotlin, una lista es una colección ordenada de elementos que puede contener elementos repetidos. Una lista se puede crear utilizando la clase `List`, que es una interfaz que define operaciones para acceder a los elementos de la lista.

En Kotlin, las listas son inmutables por defecto, lo que significa que una vez creada una lista, no se puede agregar, eliminar o modificar elementos de la misma. Si se desea crear una lista mutable, se puede utilizar la clase `MutableList`.

▼ Creación de listas en Kotlin

Para crear una lista en Kotlin, se puede utilizar la función `listOf()` que acepta cualquier número de elementos y devuelve una lista inmutable:

```
val lista = listOf("elemento1", "elemento2", "elemento3")
```

También se puede crear una lista mutable utilizando la clase `MutableList` y la función `mutableListOf()`, que acepta cualquier número de elementos y devuelve una lista mutable:

```
val listaMutable = mutableListOf("elemento1", "elemento2", "elemento3")
```

> `MutableList` es una interfaz que define las operaciones que se pueden realizar en una lista mutable, mientras que `mutableListOf()`
> es una función que crea una instancia de una lista mutable con elementos iniciales. Ambas opciones se pueden utilizar para crear una lista mutable en Kotlin, pero `mutableListOf()` es una forma más sencilla de crear una lista mutable con elementos iniciales.

▼ Accediendo a los elementos de una lista

Para acceder a los elementos de una lista en Kotlin, puedes usar el operador de indexación `[]` junto con el índice del elemento que deseas acceder. El índice comienza en cero para el primer elemento de la lista y aumenta en uno para cada

elemento subsiguiente. Por ejemplo, si tienes una lista de enteros llamada `numeros`, puedes acceder al primer elemento de la lista de la siguiente manera:

```
val numeros = listOf(1, 2, 3, 4, 5)
val primerNumero = numeros[0]
```

En este ejemplo, el primer elemento de la lista es 1, por lo que `primerNumero` será igual a 1.

También puedes acceder al último elemento de la lista usando el índice `-1`, como se muestra a continuación:

```
val ultimoNumero = numeros[numeros.size - 1]
```

En este ejemplo, `numeros.size` devuelve el tamaño de la lista, que es 5. Restando 1, obtenemos el índice del último elemento de la lista, que es 4.

▼ Modificando los elementos de una lista

Para modificar los elementos de una lista en Kotlin, puedes usar el operador de indexación `[]` junto con el índice del elemento que deseas modificar. Por ejemplo, si tienes una lista de enteros llamada `numeros` y quieres cambiar el segundo elemento a 10, puedes hacer lo siguiente:

```
val numeros = mutableListOf(1, 2, 3, 4, 5)
numeros[1] = 10
```

En este ejemplo, la lista `numeros` se define como una lista modificable usando la función `mutableListOf`, y luego se modifica el segundo elemento de la lista usando el operador de indexación `[]`.

También puedes usar la función `set` para modificar un elemento en una lista. La función `set` toma dos argumentos: el índice del elemento que deseas modificar y el nuevo valor que deseas asignar al elemento. Por ejemplo:

```
val numeros = mutableListOf(1, 2, 3, 4, 5)
numeros.set(1, 10)
```

En este ejemplo, se modifica el segundo elemento de la lista `numeros` usando la función `set`.

Ten en cuenta que si intentas modificar un índice que está fuera del rango de la lista, se producirá una excepción de tipo `IndexOutOfBoundsException`. Por lo tanto, es importante asegurarse de que el índice esté dentro del rango de la lista antes de modificar un elemento

▼ Recorriendo una lista

Hay varias formas de recorrer una lista en Kotlin. Una forma común es usar un bucle `for` que recorre la lista elemento por elemento. Puedes hacer esto de la siguiente manera:

```
val numeros = listOf(1, 2, 3, 4, 5)
for (numero in numeros) {
    println(numero)
}
```

En este ejemplo, se define una lista de enteros llamada `numeros`, y luego se recorre la lista usando un bucle `for`. En cada iteración del bucle, la variable `numero` se establece en el siguiente elemento de la lista, y se imprime el valor de la variable `numero`.

Además, puedes usar la función `forEach` para recorrer una lista y aplicar una función a cada elemento. Por ejemplo:

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

```
    println(numero)
}
```

En este ejemplo, se usa la función `forEach` para recorrer la lista `numeros` y aplicar la función lambda `{ numero -> println(numero) }` a cada elemento de la lista. La función lambda toma un argumento `numero` que representa cada elemento de la lista, y luego imprime el valor del argumento.

▼ Funciones útiles para trabajar con listas en Kotlin

Kotlin proporciona una amplia variedad de funciones útiles para trabajar con listas. Algunas de las funciones más comunes son las siguientes:

1. `size`: devuelve el número de elementos en la lista.

```
javaCopy code
val numeros = listOf(1, 2, 3, 4, 5)
val cantidad = numeros.size // 5
```

2. `isEmpty`: devuelve `true` si la lista está vacía.

```
scssCopy code
val numeros = listOf(1, 2, 3, 4, 5)
val vacia = numeros.isEmpty() // false
```

3. `contains`: devuelve `true` si la lista contiene un elemento determinado.

```
scssCopy code
val numeros = listOf(1, 2, 3, 4, 5)
val contieneTres = numeros.contains(3) // true
```

4. `indexOf`: devuelve el índice de la primera aparición de un elemento determinado en la lista, o `1` si el elemento no se encuentra en la lista.

```
scssCopy code
val numeros = listOf(1, 2, 3, 4, 5)
val indiceDeTres = numeros.indexOf(3) // 2
```

5. `lastIndexOf`: devuelve el índice de la última aparición de un elemento determinado en la lista, o `1` si el elemento no se encuentra en la lista.

```
scssCopy code
val numeros = listOf(1, 2, 3, 4, 5, 3)
val ultimoIndiceDeTres = numeros.lastIndexOf(3) // 5
```

6. `subList`: devuelve una sublista que contiene un rango de elementos de la lista.

```
scssCopy code
val numeros = listOf(1, 2, 3, 4, 5)
val sublista = numeros.subList(1, 4) // [2, 3, 4]
```

7. `sorted`: devuelve una lista ordenada en orden ascendente o descendente.

```
scssCopy code
val numeros = listOf(3, 2, 5, 1, 4)
val ascendente = numeros.sorted() // [1, 2, 3, 4, 5]
val descendente = numeros.sortedDescending() // [5, 4, 3, 2, 1]
```

8. `reversed`: devuelve una lista en orden inverso.

```
scssCopy code
val numeros = listOf(1, 2, 3, 4, 5)
val invertida = numeros.reversed() // [5, 4, 3, 2, 1]
```

Estas son solo algunas de las funciones disponibles para trabajar con listas en Kotlin. Existen muchas otras funciones que pueden resultar útiles en diferentes situaciones, por lo que se recomienda revisar la documentación oficial de Kotlin para conocer todas las funciones disponibles.

▼ Diferencias de una Lista y un Arreglo

Las principales diferencias entre ellas son las siguientes:

1. Tamaño: un arreglo tiene un tamaño fijo que se define al crearlo, mientras que una lista puede crecer o disminuir de tamaño dinámicamente a medida que se agregan o eliminan elementos.
2. Mutabilidad: un arreglo puede ser mutable o inmutable. En un arreglo mutable, se pueden cambiar los elementos individuales después de crear el arreglo. En cambio, un arreglo inmutable no se puede cambiar después de crearlo. Por otro lado, una lista es siempre mutable, lo que significa que se pueden agregar, eliminar o cambiar elementos en cualquier momento.
3. Acceso a elementos: en un arreglo, se puede acceder a los elementos mediante su índice. En cambio, en una lista, se puede acceder a los elementos mediante sus métodos, como `get` o `[]`.
4. Rendimiento: en general, los arreglos son más rápidos que las listas para acceder a elementos individuales. Sin embargo, las listas son más eficientes en cuanto a memoria y rendimiento al agregar o eliminar elementos en el medio de la estructura.

En resumen, un arreglo es más adecuado si se necesita una estructura de datos de tamaño fijo y de acceso rápido a elementos individuales, mientras que una lista es más adecuada si se necesita una estructura de datos dinámica y mutable para agregar o eliminar elementos en cualquier momento.

▼ Conjuntos en Kotlin

▼ ¿Qué es un conjunto?

En Kotlin, los conjuntos se pueden crear utilizando la clase `Set`. Hay dos tipos de conjuntos en Kotlin: `Set` y `MutableSet`. Un `Set` es una colección de solo lectura que no se puede modificar después de la creación, mientras que un `MutableSet` es una colección que se puede modificar y actualizar en cualquier momento.

Los conjuntos en Kotlin tienen algunas características importantes:

- Los elementos en un conjunto son únicos y no tienen un orden específico.
- Los conjuntos en Kotlin no permiten elementos duplicados.
- Los conjuntos se pueden usar para realizar operaciones de conjunto, como la unión, la intersección y la diferencia de conjuntos.
- Los conjuntos en Kotlin se pueden iterar utilizando un bucle `for` o métodos de iteración como `forEach()`.

En resumen, los conjuntos son una estructura de datos importante en programación que se utilizan para almacenar elementos únicos y realizar operaciones de conjunto de manera eficiente.

▼ Creación de conjuntos en Kotlin

En Kotlin, se pueden crear conjuntos utilizando la clase `Set` y la clase `MutableSet`. Hay varias formas de crear conjuntos en Kotlin, entre ellas:

1. Crear un conjunto vacío y agregar elementos individualmente:

```
kotlinCopy code
val mySet = mutableSetOf<String>()
mySet.add("elemento1")
mySet.add("elemento2")
mySet.add("elemento3")
```

2. Crear un conjunto con elementos iniciales utilizando la función `setOf()`:

```
kotlinCopy code
val mySet = setOf("elemento1", "elemento2", "elemento3")
```

3. Crear un conjunto mutable con elementos iniciales utilizando la función `mutableSetOf()`:

```
kotlinCopy code
val mySet = mutableSetOf("elemento1", "elemento2", "elemento3")
```

Una vez creado un conjunto, se pueden agregar, eliminar y modificar elementos según sea necesario. Los conjuntos también tienen una serie de funciones útiles para realizar operaciones de conjunto, como `union()`, `intersect()`, `subtract()` y `contains()`, entre otras.

▼ Accediendo a los elementos de un conjunto

En Kotlin, se pueden acceder a los elementos de un conjunto utilizando varios métodos y operaciones. Algunos de los métodos y operaciones más comunes para acceder a los elementos de un conjunto son los siguientes:

1. Utilizando el operador `in`:

```
val mySet = setOf("elemento1", "elemento2", "elemento3")
if ("elemento1" in mySet) {
    println("El conjunto contiene el elemento1")
}
```

2. Utilizando el método `contains()`:

```
val mySet = setOf("elemento1", "elemento2", "elemento3")
if (mySet.contains("elemento1")) {
    println("El conjunto contiene el elemento1")
}
```

3. Recorriendo el conjunto con un bucle `for`:

```
val mySet = setOf("elemento1", "elemento2", "elemento3")
for (elemento in mySet) {
    println(elemento)
}
```

4. Convertir el conjunto a una lista y acceder a los elementos de la lista:

```
val mySet = setOf("elemento1", "elemento2", "elemento3")
val myList = mySet.toList()
println(myList[0]) // Imprime "elemento1"
```

Recuerda que los conjuntos en Kotlin no tienen un orden específico, por lo que no es posible acceder a los elementos de un conjunto mediante un índice numérico. Si necesitas acceder a los elementos de un conjunto en un orden específico, es posible convertir el conjunto a una lista o a otra estructura de datos que admita un ordenamiento.

▼ Modificando los elementos de un conjunto

Los conjuntos (Sets) son colecciones de elementos únicos, lo que significa que no se pueden modificar los elementos de un conjunto directamente. Si deseas modificar los elementos de un conjunto, debes eliminar el elemento original y agregar un nuevo elemento.

Por ejemplo, supongamos que tienes un conjunto de nombres:

```
val nombres = mutableSetOf("Ana", "Juan", "María")
```

Para modificar el nombre "Juan" a "Pedro", primero debes eliminar "Juan" del conjunto y luego agregar "Pedro":

```
nombres.remove("Juan")
nombres.add("Pedro")
```

Ahora, si imprimes el conjunto de nombres, verás que el nombre "Juan" ha sido reemplazado por "Pedro":

```
println(nombres) // Output: [Ana, Pedro, María]
```

Ten en cuenta que, si intentas agregar un elemento que ya existe en el conjunto, no se agregará ningún elemento nuevo y el conjunto seguirá siendo el mismo.

```
nombres.add("Ana")
println(nombres) // Output: [Ana, Pedro, María]
```

▼ Recorriendo un conjunto

Se puede recorrer un conjunto usando diferentes tipos de bucles y funciones.

1. Bucle for

   Puedes usar un `Bucle For` para recorrer todos los elementos de un conjunto. Aquí un ejemplo:

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

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

2. Función forEach:

   La función `forEach` te permite recorrer un conjunto y realizar una acción en cada elemento. Aquí un ejemplo:

```
val frutas = setOf("Manzana", "Banana", "Naranja")

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

3. Función forEachIndexed:

   La función `forEachIndexed` te permite recorrer un conjunto y realizar una acción en cada elemento junto con su índice. Aquí un ejemplo:

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

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

En resumen, hay varias formas de recorrer un conjunto en Kotlin. Puedes usar un bucle for, la función `forEach` o la función `forEachIndexed`, dependiendo de tus necesidades.

▼ Funciones útiles para trabajar con conjuntos en Kotlin

hay varias funciones útiles para trabajar con conjuntos (Sets). Aquí te presento algunas de ellas:

1. `add(element: E)`: Agrega un elemento al conjunto.

```
val conjunto = mutableSetOf("manzana", "banana")
conjunto.add("naranja")
```

2. `addAll(elements: Collection<E>)`: Agrega una colección de elementos al conjunto.

```
val conjunto = mutableSetOf("manzana", "banana")
val frutas = listOf("naranja", "pera")
conjunto.addAll(frutas)
```

3. `remove(element: E)`: Elimina un elemento del conjunto.

```
val conjunto = mutableSetOf("manzana", "banana")
conjunto.remove("manzana")
```

4. `removeAll(elements: Collection<E>)`: Elimina una colección de elementos del conjunto.

```
val conjunto = mutableSetOf("manzana", "banana", "naranja", "pera")
val frutas = listOf("naranja", "pera")
conjunto.removeAll(frutas)
```

5. `retainAll(elements: Collection<E>)`: Elimina todos los elementos del conjunto que no están en la colección especificada.

```
val conjunto = mutableSetOf("manzana", "banana", "naranja", "pera")
val frutas = listOf("naranja", "pera")
conjunto.retainAll(frutas)
```

6. `contains(element: E)`: Verifica si el conjunto contiene un elemento específico.

```
val conjunto = setOf("manzana", "banana")
val contieneManzana = conjunto.contains("manzana") // true
val contieneNaranja = conjunto.contains("naranja") // false
```

7. `size`: Devuelve el número de elementos en el conjunto.

```
val conjunto = setOf("manzana", "banana")
val tamaño = conjunto.size // 2
```

8. `isEmpty`: Verifica si el conjunto está vacío.

```
val conjuntoVacio = setOf<String>()
val conjuntoConElementos = setOf("manzana", "banana")
val estaVacio1 = conjuntoVacio.isEmpty() // true
val estaVacio2 = conjuntoConElementos.isEmpty() // false
```

9. `intersect(other: Set<E>): Set<E>`: Devuelve un conjunto que contiene los elementos que están en ambos conjuntos.

```
val conjunto1 = setOf("manzana", "banana", "naranja")
val conjunto2 = setOf("banana", "naranja", "pera")
val intersección = conjunto1.intersect(conjunto2) // {banana, naranja}
```

10. `union(other: Set<E>): Set<E>`: Devuelve un conjunto que contiene todos los elementos de ambos conjuntos.

```
val conjunto1 = setOf("manzana", "banana")
val conjunto2 = setOf("naranja", "pera")
val unión = conjunto1.union(conjunto2) // {manzana, banana, naranja, pera}
```

Estas son solo algunas de las funciones útiles para trabajar con conjuntos en Kotlin. Hay muchas más funciones disponibles en la API de Kotlin, y también puedes crear tus propias funciones personalizadas según tus necesidades.

▼ Mapas en Kotlin

▼ ¿Qué es un mapa?

Es una estructura de datos que almacena elementos en forma de pares clave-valor. Cada elemento del mapa tiene una clave única que se utiliza para acceder al valor asociado a esa clave. Los mapas son muy útiles cuando se necesita almacenar y acceder a información por medio de una clave. Por ejemplo, podríamos tener un mapa que contenga el precio de cada producto en una tienda, donde la clave sería el nombre del producto y el valor sería el precio.

▼ Creación de mapas en Kotlin

En Kotlin, se puede crear un mapa utilizando la interfaz `Map`. Hay dos tipos de mapas en Kotlin:

1. `Map<K, V>`: Es un mapa de sólo lectura, donde K es el tipo de la clave y V es el tipo del valor.
2. `MutableMap<K, V>`: Es un mapa que se puede modificar, donde K es el tipo de la clave y V es el tipo del valor.

Aquí te presento un ejemplo de cómo crear un mapa en Kotlin:

```
// Crear un mapa de sólo lectura
val mapa1 = mapOf("manzana" to 1.99, "banana" to 0.99, "naranja" to 1.49)

// Crear un mapa mutable
val mapa2 = mutableMapOf("manzana" to 1.99, "banana" to 0.99, "naranja" to 1.49)

// Agregar un nuevo elemento al mapa mutable
mapa2["pera"] = 2.49
```

En este ejemplo, `mapa1` es un mapa de sólo lectura que contiene tres elementos, donde cada clave es el nombre de una fruta y el valor es su precio. `mapa2` es un mapa mutable que contiene los mismos elementos que `mapa1`, pero también se agrega un nuevo elemento con la clave "pera" y el valor 2.49.

Los mapas son muy útiles en muchos casos, como por ejemplo, para almacenar configuraciones, para hacer un mapeo de nombres a direcciones de correo electrónico, o para almacenar información que se utilizará frecuentemente y que se accederá con una clave específica.

▼ Accediendo a los elementos de un mapa

Para acceder a los elementos de un mapa se puede utilizar la clave correspondiente. Por ejemplo, si tenemos un mapa que contiene el precio de algunas frutas, podemos acceder al precio de una fruta en particular utilizando la clave que se le ha asignado:

```
val preciosFrutas = mapOf("manzana" to 1.99, "banana" to 0.99, "naranja" to 1.49)

val precioManzana = preciosFrutas["manzana"] // Devuelve 1.99
val precioBanana = preciosFrutas["banana"] // Devuelve 0.99
val precioPera = preciosFrutas["pera"] // Devuelve null, ya que "pera" no está en el mapa
```

En este ejemplo, `preciosFrutas` es un mapa que contiene los precios de algunas frutas. La variable `precioManzana` obtiene el precio de la manzana accediendo al elemento correspondiente del mapa utilizando la clave `"manzana"`. De forma similar, `precioBanana` obtiene el precio de la banana y `precioPera` intenta obtener el precio de la pera, pero devuelve `null` porque esa clave no está en el mapa.

También se puede utilizar el método `get` para obtener el valor de un elemento del mapa:

```
val precioManzana = preciosFrutas.get("manzana") // Devuelve 1.99
```

Si el mapa no contiene la clave que se está buscando, se puede utilizar el método `getOrDefault` para obtener un valor predeterminado en su lugar:

```
val precioPera = preciosFrutas.getOrDefault("pera", 2.49) // Devuelve 2.49, ya que "pera" no está en el mapa
```

En este caso, el valor predeterminado es `2.49`, que es el precio que se asigna a la pera en caso de que no se encuentre en el mapa.

▼ Modificando los elementos de un mapa

Se puede modificar los elementos de un mapa mutable utilizando la clave correspondiente.

Por ejemplo, supongamos que tenemos un mapa mutable que contiene los precios de algunas frutas:

```
val preciosFrutas = mutableMapOf("manzana" to 1.99, "banana" to 0.99, "naranja" to 1.49)
```

Podemos modificar el precio de una fruta en particular utilizando la clave que se le ha asignado y el operador de asignación `=`:

```
preciosFrutas["manzana"] = 2.49 // Modifica el precio de la manzana a 2.49
```

En este ejemplo, la línea de código modifica el precio de la manzana en el mapa, cambiándolo de 1.99 a 2.49.

También se puede utilizar el método `put` para agregar o actualizar un elemento en el mapa mutable:

```
preciosFrutas.put("pera", 2.99) // Agrega el precio de la pera al mapa
preciosFrutas.put("naranja", 1.99) // Actualiza el precio de la naranja en el mapa
```

En este ejemplo, la primera línea de código agrega el precio de la pera al mapa mutable. La segunda línea de código actualiza el precio de la naranja en el mapa mutable, cambiándolo de 1.49 a 1.99.

También se puede utilizar el método `putIfAbsent` para agregar un elemento al mapa sólo si la clave no existe:

```
preciosFrutas.putIfAbsent("manzana", 2.99) // No modifica el precio de la manzana, ya que la clave ya existe
preciosFrutas.putIfAbsent("kiwi", 3.49) // Agrega el precio del kiwi al mapa
```

En este ejemplo, la primera línea de código no modifica el precio de la manzana, ya que la clave `"manzana"` ya existe en el mapa mutable. La segunda línea de código agrega el precio del kiwi al mapa mutable, ya que la clave `"kiwi"` no existe en el mapa.

▼ Recorriendo un mapa

Se puede recorrer un mapa utilizando un bucle `for` y la función `forEach`. La función `forEach` se utiliza para iterar sobre los elementos del mapa y ejecutar una acción en cada uno de ellos.

Por ejemplo, supongamos que tenemos un mapa que contiene los precios de algunas frutas:

```
val preciosFrutas = mapOf("manzana" to 1.99, "banana" to 0.99, "naranja" to 1.49)
```

Podemos recorrer el mapa utilizando un bucle `for` y la función `forEach` para imprimir los precios de todas las frutas:

```
preciosFrutas.forEach { (fruta, precio) ->
    println("$fruta cuesta $precio")
}
```

En este ejemplo, la función `forEach` recibe una función lambda que se ejecutará en cada elemento del mapa. La función lambda toma dos parámetros: la clave y el valor del elemento. En este caso, utilizamos la sintaxis de desestructuración para separar la clave y el valor en dos variables distintas: `fruta` y `precio`. Luego, utilizamos la función `println` para imprimir el nombre de la fruta y su precio.

El resultado de la ejecución del código sería el siguiente:

```
manzana cuesta 1.99
banana cuesta 0.99
naranja cuesta 1.49
```

También se puede utilizar un bucle `for` para recorrer un mapa y acceder a cada elemento utilizando su clave:

```
for (fruta in preciosFrutas.keys) {
    val precio = preciosFrutas[fruta]
```

```
    println("$fruta cuesta $precio")
}
```

En este ejemplo, utilizamos el método `keys` para obtener un conjunto con las claves del mapa. Luego, utilizamos un bucle `for` para iterar sobre cada clave y acceder al valor correspondiente utilizando el operador de indexación `[]`. Por último, utilizamos la función `println` para imprimir el nombre de la fruta y su precio.

▼ Funciones útiles para trabajar con mapas en Kotlin

Algunas de las funciones más comunes son:

1. `get`: se utiliza para obtener el valor correspondiente a una clave específica. Si la clave no existe, devuelve `null`. Por ejemplo:

```
val preciosFrutas = mapOf("manzana" to 1.99, "banana" to 0.99, "naranja" to 1.49)
val precioManzana = preciosFrutas.get("manzana")
val precioKiwi = preciosFrutas.get("kiwi")
```

   En este ejemplo, `precioManzana` sería igual a `1.99`, que es el valor correspondiente a la clave `"manzana"`. `precioKiwi` sería igual a `null`, ya que la clave `"kiwi"` no existe en el mapa.

2. `getOrElse`: se utiliza para obtener el valor correspondiente a una clave específica, pero si la clave no existe, se devuelve un valor predeterminado. Por ejemplo:

```
val precioPera = preciosFrutas.getOrElse("pera") { 2.99 }
val precioKiwi = preciosFrutas.getOrElse("kiwi") { 3.49 }
```

   En este ejemplo, `precioPera` sería igual a `2.99`, que es el valor predeterminado que se ha especificado. `precioKiwi` sería igual a `3.49`, ya que la clave `"kiwi"` no existe en el mapa y se ha especificado un valor predeterminado.

3. `containsKey`: se utiliza para comprobar si una clave específica existe en el mapa. Devuelve `true` si la clave existe, y `false` en caso contrario. Por ejemplo:

```
val tieneManzana = preciosFrutas.containsKey("manzana")
val tieneKiwi = preciosFrutas.containsKey("kiwi")
```

   En este ejemplo, `tieneManzana` sería igual a `true`, ya que la clave `"manzana"` existe en el mapa. `tieneKiwi` sería igual a `false`, ya que la clave `"kiwi"` no existe en el mapa.

4. `keys`: se utiliza para obtener un conjunto con las claves del mapa. Por ejemplo:

```
val claves = preciosFrutas.keys
```

   En este ejemplo, `claves` sería un conjunto que contiene las claves `"manzana"`, `"banana"` y `"naranja"`.

5. `values`: se utiliza para obtener una colección con los valores del mapa. Por ejemplo:

```
val valores = preciosFrutas.values
```

   En este ejemplo, `valores` sería una colección que contiene los valores `1.99`, `0.99` y `1.49`.

6. `filter`: se utiliza para filtrar los elementos del mapa según un predicado específico. Devuelve un nuevo mapa que contiene sólo los elementos que cumplen el predicado. Por ejemplo:

```
val preciosAltos = preciosFrutas.filter { (_, precio) -> precio > 1.5 }
```

En este ejemplo, la función `filter` devuelve un nuevo mapa que contiene sólo los elementos cuyo precio es mayor que `1.5`. El nuevo mapa tendría la siguiente apariencia: `{"manzana" to 1.99, "naranja" to 1.49}`.

▼ Pares en Kotlin

▼ ¿Qué es un par?

Es un tipo de datos que representa un conjunto ordenado de dos elementos. Es decir, un par es una estructura que contiene dos valores, uno de un tipo de datos y otro de otro tipo de datos, que se pueden acceder utilizando las propiedades `first` y `second`.

▼ Creación de paren en Kotlin

La sintaxis para crear un par es la siguiente:

```
val miPar = Pair(valor1, valor2)
```

donde `valor1` y `valor2` son los dos valores que se desean almacenar en el par. También se puede utilizar la notación de infix para crear un par:

```
val miPar = valor1 to valor2
```

donde `valor1` y `valor2` son los dos valores que se desean almacenar en el par, y `to` es una función de infix que crea el par.

> Los pares son útiles cuando se necesita pasar o devolver múltiples valores a una función o método, ya que sólo se puede devolver un valor en Kotlin. También son útiles para almacenar temporalmente dos valores relacionados, como una coordenada x e y en un plano cartesiano, o un nombre y una edad.

▼ Accediendo a los elementos de un par

Para acceder a los elementos de un par en Kotlin, se utilizan las propiedades `first` y `second`. Estas propiedades devuelven el primer y segundo elemento del par, respectivamente.

Por ejemplo, si tenemos el siguiente par:

```
val miPar = Pair("Hola", 123)
```

Podemos acceder a sus elementos de la siguiente manera:

```
val primerElemento = miPar.first  // "Hola"
val segundoElemento = miPar.second  // 123
```

También se puede utilizar la desestructuración de pares para asignar los valores a variables individuales:

```
val (primerElemento, segundoElemento) = miPar
```

En este caso, la variable `primerElemento` sería igual a `"Hola"`, y la variable `segundoElemento` sería igual a `123`.

▼ Modificando los elementos de un par

Los pares son inmutables, lo que significa que una vez que se han creado, no se pueden modificar sus valores.

Para cambiar los valores de un par, se debe crear un nuevo par con los valores actualizados. Por ejemplo:

```
val miPar = Pair("Hola", 123)
val nuevoPar = Pair("Adiós", 456)
```

En este caso, no es posible modificar los valores del par `miPar`. Si se desea cambiar los valores, se debe crear un nuevo par, como se hizo con `nuevoPar`.

También es posible utilizar la función `copy` para crear una copia del par con uno o ambos elementos actualizados. La función `copy` devuelve un nuevo par con los mismos valores que el original, excepto por los elementos actualizados. Por ejemplo:

```
val miPar = Pair("Hola", 123)
val nuevoPar = miPar.copy(first = "Adiós")
```

En este caso, `nuevoPar` sería un nuevo par con el primer elemento actualizado a `"Adiós"`, mientras que el segundo elemento seguiría siendo `123`.

▼ Recorriendo un par

Al ser un tipo de dato que contiene solamente dos elementos, no es necesario recorrer un par en Kotlin. Simplemente se puede acceder a los elementos del par utilizando las propiedades `first` y `second`, como se muestra en el siguiente ejemplo:

```
val miPar = Pair("Hola", 123)
println(miPar.first)   // Imprime "Hola"
println(miPar.second)  // Imprime 123
```

También se puede utilizar la desestructuración de pares para acceder a los elementos individualmente:

```
val (primerElemento, segundoElemento) = miPar
println(primerElemento)   // Imprime "Hola"
println(segundoElemento)  // Imprime 123
```

En resumen, no es necesario recorrer un par en Kotlin, ya que se puede acceder a sus elementos directamente a través de sus propiedades o mediante la desestructuración de pares.

▼ Funciones útiles para trabajar con pares en Kotlin

En Kotlin, existen varias funciones útiles para trabajar con pares:

1. `to`: Esta función se utiliza para crear pares. Por ejemplo:

   ```
   val miPar = "Hola" to 123
   ```

   En este caso, `miPar` sería un par que contiene el string "Hola" como primer elemento y el número 123 como segundo elemento.

2. `first` y `second`: Estas son las propiedades de un par que se utilizan para acceder a su primer y segundo elemento, respectivamente. Por ejemplo:

```
val miPar = Pair("Hola", 123)
println(miPar.first)   // Imprime "Hola"
println(miPar.second)  // Imprime 123
```

3. `component1` y `component2`: Estas son las funciones que se utilizan para acceder al primer y segundo elemento de un par, respectivamente. Son equivalentes a las propiedades `first` y `second`. Por ejemplo:

```
val miPar = Pair("Hola", 123)
println(miPar.component1())   // Imprime "Hola"
println(miPar.component2())   // Imprime 123
```

5. `copy`: Esta función se utiliza para crear una copia de un par con uno o ambos elementos actualizados. Por ejemplo:

```
val miPar = Pair("Hola", 123)
val nuevoPar = miPar.copy(first = "Adiós")
```

En este caso, `nuevoPar` sería un nuevo par con el primer elemento actualizado a `"Adiós"`, mientras que el segundo elemento seguiría siendo `123`.

6. Desestructuración de pares: Esta es una técnica que se utiliza para asignar los elementos de un par a variables individuales. Por ejemplo:

```
val miPar = Pair("Hola", 123)
val (primerElemento, segundoElemento) = miPar
println(primerElemento)   // Imprime "Hola"
println(segundoElemento)  // Imprime 123
```

En este caso, la variable `primerElemento` sería igual a `"Hola"`, y la variable `segundoElemento` sería igual a `123`.

▼ Prácticas de estructuras de datos en Kotlin

▼ Ejercicios prácticos de datos en Kotlin

1. Arreglos:
   a. Dado un arreglo de calificaciones numéricas, escribir una función que calcule y retorne el promedio de dichas calificaciones.
   b. Dado un arreglo de cadenas, escribir una función que lo ordene alfabéticamente y retorne el arreglo ordenado.
2. Listas:
   a. Dada una lista de números enteros, escribir una función que filtre y retorne una lista con los números pares de la lista original.
   b. Dada una lista de cadenas, escribir una función que elimine los elementos duplicados y retorne una nueva lista sin duplicados.
3. Conjuntos:
   a. Dados dos conjuntos, escribir una función que retorne un nuevo conjunto que contenga la unión de ambos conjuntos.
   b. Dados dos conjuntos, escribir una función que retorne un nuevo conjunto que contenga los elementos que están en el primer conjunto, pero no en el segundo conjunto.

4. Mapas:
    a. Dada una lista de elementos, escribir una función que retorne un mapa que cuente cuántas veces aparece cada elemento en la lista
    b. Acceder y recorrer un mapa en Kotlin.

5. Pares:
    a. Dada una lista de pares (cada par contiene dos números), escribir una función que encuentre y retorne el par más grande en la lista (es decir, el par cuyos dos números suman el valor más grande).
    b. Asignación y acceso a valores de pares en Kotlin

▼ Solución a los ejercicios prácticos

1. Arreglos:

    a.

```
// Esta función toma un arreglo de enteros como parámetro y calcula su promedio
fun calcularPromedio(calificaciones: IntArray): Double {
    var sumatoria = 0
    for (calificacion in calificaciones) {
        sumatoria += calificacion
    }
    return sumatoria.toDouble() / calificaciones.size
}


fun main() {

    // Creamos un arreglo de calificaciones
    val calificaciones = intArrayOf(8, 7, 9, 10, 8)

    // Llamamos a la función "calcularPromedio" pasándole como parámetro el arreglo de calificaciones
    val promedio = calcularPromedio(calificaciones)

    // Imprimimos el resultado en consola
    println("El promedio de calificaciones es: $promedio")
}
```

    b.

```
fun ordenarCadenas(cadenas: Array<String>): Array<String> {
    //la función sortedArray() se utiliza para ordenar el arreglo alfabéticamente.
    return cadenas.sortedArray()
}

fun main(){
    val palabras = arrayOf("manzana", "pera", "banana", "kiwi")
    val palabrasOrdenadas = ordenarCadenas(palabras)

    //La función contentToString() se utiliza para imprimir el arreglo ordenado como una cadena.
    println("Palabras ordenadas: ${palabrasOrdenadas.contentToString()}")
}
```

2. Listas:

    a.

```
// La función recibe una lista de números enteros y retorna una lista con los números pares.
fun filtrarPares(numeros: List<Int>): List<Int> {

    // La lista original es filtrada y solo se retornan los números pares.
    return numeros.filter { it % 2 == 0 }
}

fun main(){
```

```
    val numeros = listOf(3, 7, 4, 8, 2, 1)

    // Se llama a la función filtrarPares con la lista de números.
    val numerosPares = filtrarPares(numeros)

    // Se imprime en consola la lista de números pares.
    println("Números pares: $numerosPares")
}
```

b.

```
// Definición de la función eliminarDuplicados, que recibe una lista de cadenas y devuelve otra lista sin duplicados.
fun eliminarDuplicados(cadenas: List<String>): List<String> {
return cadenas.distinct()
}

// Función main que crea una lista de frutas con duplicados, llama a la función eliminarDuplicados y muestra el resultado.
fun main(){
val frutas = listOf("manzana", "naranja", "manzana", "kiwi", "naranja")
val frutasSinDuplicados = eliminarDuplicados(frutas)
println("Frutas sin duplicados: $frutasSinDuplicados")
}
```

3. Conjuntos:

   a.

```
// Función que recibe dos conjuntos de números enteros y retorna su unión
fun unirConjuntos(conjunto1: Set<Int>, conjunto2: Set<Int>): Set<Int> {

    // Utiliza la función "union" de la clase Set para obtener la unión de los conjuntos
    return conjunto1.union(conjunto2)
}

// Función principal del programa
fun main() {
    // Crea un conjunto con los números 1, 2, 3 y 4
    val conjunto1 = setOf(1, 2, 3, 4)

    // Crea otro conjunto con los números 3, 4, 5 y 6
    val conjunto2 = setOf(3, 4, 5, 6)

    // Llama a la función "unirConjuntos" para obtener la unión de los conjuntos anteriores
    val union = unirConjuntos(conjunto1, conjunto2)

    // Imprime en consola el resultado de la unión de los conjuntos
    println("Unión de conjuntos: $union")
}
```

b.

```
// Función que recibe dos conjuntos de números enteros y retorna su unión
fun unirConjuntos(conjunto1: Set<Int>, conjunto2: Set<Int>): Set<Int> {

    // Utiliza la función "union" de la clase Set para obtener la unión de los conjuntos
    return conjunto1.union(conjunto2)
}

// Función principal del programa
fun main() {
    // Crea un conjunto con los números 1, 2, 3 y 4
    val conjunto1 = setOf(1, 2, 3, 4)

    // Crea otro conjunto con los números 3, 4, 5 y 6
    val conjunto2 = setOf(3, 4, 5, 6)

    // Llama a la función "unirConjuntos" para obtener la unión de los conjuntos anteriores
    val union = unirConjuntos(conjunto1, conjunto2)

    // Imprime en consola el resultado de la unión de los conjuntos
```

```
    println("Unión de conjuntos: $union")
}
```

4. Mapas:

    a.

```
// Esta función recibe una lista de cadenas y devuelve un mapa que asocia a cada cadena su cantidad de ocurrencias en la lista
fun contarElementos(lista: List<String>): Map<String, Int> {
    // La función groupingBy agrupa los elementos de la lista según una función que le pasamos, que en este caso es simplement
    // Esto nos da un objeto Grouping que podemos usar para contar las ocurrencias de cada elemento con la función eachCount.
    return lista.groupingBy { it }.eachCount()
}

fun main(){
    // Creamos una lista con algunas cadenas repetidas.
    val lista = listOf("manzana", "banana", "naranja", "manzana", "kiwi", "banana", "manzana")
    // Contamos los elementos de la lista usando la función contarElementos que definimos anteriormente.
    val conteo = contarElementos(lista)
    // Imprimimos el resultado.
    println("Conteo de elementos: $conteo")
}
```

    b.

```
// Se define un mapa de nombre y edad
val mapaEdades = mapOf(
    "Juan" to 25,
    "María" to 30,
    "Pedro" to 20,
    "Ana" to 28,
    "Luis" to 35
)

// Se accede a los valores del mapa usando las claves
println("La edad de Juan es ${mapaEdades["Juan"]} años")
println("La edad de Ana es ${mapaEdades["Ana"]} años")

// Se recorre el mapa usando un bucle for
for ((nombre, edad) in mapaEdades) {
    println("$nombre tiene $edad años")
}
```

5. Pares:

    a.

```
fun parMasGrande(lista: List<Pair<Int, Int>>): Pair<Int, Int>? {
    return lista.maxByOrNull { it.first + it.second }
}

fun main(){

    // Creamos una lista de pares de números enteros
    val lista = listOf(
        Pair(1, 2),
        Pair(3, 4),
        Pair(5, 6),
        Pair(7, 8),
        Pair(2, 3)
    )

    // Llamamos a la función parMasGrande y guardamos el resultado en la variable parMaximo
    val parMaximo = parMasGrande(lista)

    // Imprimimos el resultado en la consola
    println("Par más grande: $parMaximo")

}
```

b.

```
fun main() {
    // Se crea un par que contiene una cadena y un número
    val miPar = Pair("Hola", 123)

    // Se accede a los valores del par usando sus propiedades first y second
    val cadena = miPar.first
    val numero = miPar.second

    // Se imprimen los valores
    println("La cadena es: $cadena")
    println("El número es: $numero")
}
```