Estructuras de Datos en Kotlin

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

    Son formas organizadas de almacenar y manipular datos en un programa de computadora. Se utilizan para almacenar y organizar datos de manera eficiente y permitir un acceso rápido y eficaz a los mismos. Algunas de las estructuras de datos más comunes incluyen arrays, listas enlazadas, pilas, colas, árboles y grafos.
  - Ventajas de utilizar estructuras de datos en Kotlin
    - Organización de Datos: las estructuras de datos proporciona una forma organizada de almacenar y acceder a los datos. Por ejemplo, las listas y los conjuntos proporcionan una forma de almacenar elementos en una secuencia ordenada o desordenada, mientras que los mapas proporcionan una forma de almacenar pares clave-valor.
    - Eficiencia: Dependiendo de la estructura de datos que se utilice, se pueden obtener operaciones eficientes en términos de tiempo y espacio. Por ejemplo, el acceso a elementos de una lista o un arreglo es más rápido que en una estructura de datos basada en enlaces como una lista enlazada.
    - Facilidad de uso: Kotlin proporciona varias estructuras de datos integradas que son fáciles de usar y entender. Por ejemplo, la clase MutableList permite la modificación de elementos en una lista, mientras que la clase Set garantiza la ausencia de elementos duplicados.
    - Flexibilidad: Kotlin proporciona una variedad de estructuras de datos, cada una con sus propias ventajas y desventajas. Por lo tanto, se puede seleccionar la estructura de datos que mejor se adapte a los requisitos específicos de la aplicación.
    - Operaciones de alto nivel: Kotlin proporciona una variedad de operaciones de alto nivel en sus estructuras de datos integradas. Por ejemplo, la función filter () permite filtrar elementos de una lista que cumplan con un determinado criterio, mientras que la función reduce () permite reducir una lista a un solo valor.
  - Diferencias entre las estructuras de datos en Kotlin y Java
    - Null Safety: En Kotlin, la seguridad de nulos (null safety) es una característica importante. Kotlin requiere que los tipos de datos sean explícitamente marcados como nulos o no nulos, lo que ayuda a prevenir errores de NullPointerException. Java, por otro lado, no tiene una característica de null safety incorporada.
    - Tipos de datos de colecciones: Kotlin proporciona una gran cantidad de tipos de colecciones en comparación con Java. Kotlin tiene varias colecciones útiles, como "`mutableListOf`" y "`mutableSetOf`", que no están presentes en Java.
    - Interoperabilidad con Java: Kotlin fue diseñado para ser interoperable con Java, lo que significa que las estructuras de datos en Kotlin pueden ser fácilmente usadas en Java y viceversa. Sin embargo, Kotlin tiene algunas características adicionales, como la expresión de funciones y la seguridad de nulos, que pueden requerir algunas modificaciones para hacer que las estructuras de datos en Java sean compatibles con Kotlin.
    - Constructores primarios: En Kotlin, las clases pueden tener constructores primarios y constructores secundarios, lo que permite una mayor flexibilidad en la creación de objetos. Java, por otro lado, solo admite un constructor por clase.
    - Extensiones de funciones: Kotlin admite extensiones de funciones, lo que permite agregar métodos a clases existentes sin necesidad de herencia o modificaciones en la clase original. Java no admite extensiones de funciones.
- Arreglos en Kotlin

▼ ¿Qué es un arreglo?

es una estructura de datos que se utiliza para almacenar una colección de elementos del mismo tipo. Los elementos se almacenan en posiciones consecutivas de memoria, y cada posición se identifica por un índice o subíndice que comienza en cero.

▼ Creación de arreglos en Kotlin

Usando la función `arrayOf()`: Esta función crea un arreglo y asigna valores a cada uno de sus elementos.

```
fun main() {
    var miArreglo = arrayOf(1, 2, 3, 4, 5)

println("mi arreglo $miArreglo")
}
```

▼ Accediendo a los elementos de un arreglo

Para acceder a los elementos de un arreglo en Kotlin, se utiliza el operador de índice `[]` seguido del número de índice del elemento al que se desea acceder.

```
//el numero 0 se utliza para acceder al primer elemento
//del arreglo
val primerElemento = miArreglo[0]
```

▼ Modificando los elementos de un arreglo

En Kotlin, los elementos de un arreglo pueden ser modificados mediante el operador de índice `[]` al igual que se hace para acceder a ellos. Por ejemplo, si tenemos el siguiente arreglo:

```
var miArreglo = arrayOf(1, 2, 3, 4, 5)
```

Podemos cambiar el valor del primer elemento del arreglo (el número 1) de la siguiente manera:

```
miArreglo[0] = 10
```

En este caso, se asigna el valor `10` al primer elemento del arreglo. Para poder modificar los elementos de un arreglo, es importante que la variable que lo contiene sea declarada como mutable, utilizando la palabra clave `var`.

▼ Recorriendo un arreglo

En Kotlin, existen varias formas de recorrer los elementos de un arreglo. Una de las formas más comunes es utilizar un bucle `for`, como se muestra a continuación:

```
val miArreglo = arrayOf(1, 2, 3, 4, 5)

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

En este caso, se utiliza un bucle `for` para recorrer todos los elementos del arreglo `miArreglo` e imprimirlos en la consola. La variable `elemento` toma el valor de cada elemento del arreglo en cada iteración del bucle.

▼ Funciones útiles para trabajar con arreglos en Kotlin

- **ArrayOf**: Crea un nuevo arreglo con los elementos especificados.

```
val arr = arrayOf(1, 2, 3, 4, 5)
```

- **Size**: Devuelve el tamaño del arreglo.

```
val size = arr.size
```

- **Get**: Devuelve el valor en la posición especificada del arreglo.

```
val value = arr.get(3)
```

- **Set**: Asigna un valor a la posición especificada del arreglo.

```
arr.set(2, 10)
```

- **IndexOf**: Devuelve la primera posición del elemento especificado en el arreglo, o -1 si no está presente.

```
val index = arr.indexOf(3)
```

- **SliceArray**: Devuelve un nuevo arreglo que contiene un subconjunto de elementos del arreglo original.

```
val subArray = arr.sliceArray(1..3)
```

- **Filter**: Devuelve un nuevo arreglo que contiene solo los elementos que cumplen con la condición especificada.

```
val filteredArray = arr.filter { it > 3 }
```

- **Map**: Devuelve un nuevo arreglo que contiene los resultados de aplicar una función a cada elemento del arreglo original.

```
val mappedArray = arr.map { it * 2 }
```

- **Fold**: Combina todos los elementos del arreglo en un solo valor utilizando una operación específica.

```
val sum = arr.fold(0) { acc, value -> acc + value }
```

- Sorted: Devuelve un nuevo arreglo que contiene los elementos del arreglo original en orden ascendente.

```
val sortedArray = arr.sorted()
```

▼ Listas en Kotlin

▼ ¿Qué es una lista?

En programación, una lista (también conocida como lista enlazada o linked list en inglés) es una estructura de datos lineal que consiste en una secuencia de nodos, donde cada nodo contiene un valor y un puntero o referencia al siguiente nodo en la lista. Es decir, los elementos de una lista no se almacenan en una ubicación contigua en la memoria, sino que se enlazan mediante punteros.

En una lista, cada elemento (o nodo) contiene dos partes: el valor que se quiere almacenar y una referencia al siguiente elemento en la lista. La lista comienza con un nodo inicial (o cabeza) y termina con un nodo final (o cola), donde el puntero al siguiente elemento es nulo.

Las listas son una estructura de datos muy útil porque permiten agregar y eliminar elementos de manera eficiente en cualquier punto de la lista. Sin embargo, el acceso aleatorio a los elementos de la lista puede ser menos eficiente que en una estructura de datos como un arreglo, debido a que para acceder a un elemento en una lista se debe recorrer la lista desde el nodo inicial hasta el nodo deseado.

▼ Creación de listas en Kotlin

- Utilizando la función `listOf`: Esta función crea una lista inmutable que no se puede modificar una vez creada. Los elementos de la lista se pasan como argumentos separados por comas dentro de los paréntesis.

```
val lista1 = listOf("Manzana", "Naranja", "Plátano", "Mango")
```

- Utilizando la función `mutableListOf`: Esta función crea una lista mutable que se puede modificar después de su creación. Los elementos de la lista se pasan como argumentos separados por comas dentro de los paréntesis.

```
val lista2 = mutableListOf(1, 2, 3, 4, 5)
```

- Utilizando el constructor `ArrayList`: Este constructor crea una lista mutable que se puede modificar después de su creación. Los elementos de la lista se pasan como argumentos separados por comas dentro de los paréntesis.

```
val lista3 = ArrayList<String>()
lista3.add("Perro")
lista3.add("Gato")
lista3.add("Pájaro")
```

- Utilizando la función `arrayListOf`: Esta función crea una lista mutable que se puede modificar después de su creación y acepta una cantidad variable de argumentos. También se puede crear una lista a partir de un arreglo utilizando el operador spread `*`.

```
val lista4 = arrayListOf("Rojo", "Verde", "Azul")
val lista5 = arrayListOf(*arrayOf(1, 2, 3, 4, 5))
```

▼ Accediendo a los elementos de una lista

En Kotlin, se puede acceder a los elementos de una lista utilizando el operador de indexación `[]` o el método `get()`. Por ejemplo:

```
val lista = listOf("Manzana", "Naranja", "Plátano", "Mango")
val segundoElemento = lista[1]
val tercerElemento = lista.get(2)
```

También se puede utilizar el método `subList()` para obtener una lista que contiene una porción de los elementos de la lista original. El método `subList()` toma dos argumentos: el índice de inicio y el índice de fin de la porción de la lista que se desea obtener. El índice de inicio se incluye en la porción de la lista, mientras que el índice de fin no. Por ejemplo:

```
val lista = listOf("Manzana", "Naranja", "Plátano", "Mango")
val subLista = lista.subList(1, 3)
```

Es importante recordar que las listas en Kotlin son inmutables por defecto, lo que significa que una vez creada la lista, no se pueden modificar los elementos de la lista. Si se necesita una lista mutable, se debe utilizar la interfaz `MutableList`.

▼ Modificando los elementos de una lista

Para modificar los elementos de una lista mutable, se puede utilizar el operador de indexación `[]` o el método `set()`. Por ejemplo, para cambiar el segundo elemento de una lista mutable `lista`, se puede hacer lo siguiente:

```
val lista = mutableListOf("Manzana", "Naranja", "Plátano", "Mango")
lista[1] = "Pera"
lista.set(2, "Piña")
```

También se pueden agregar y eliminar elementos de una lista mutable utilizando los métodos `add()`, `addAll()`, `remove()`, `removeAt()`, `removeAll()`, `clear()`, entre otros. Por ejemplo, para agregar un elemento al final de la lista, se puede utilizar el método `add()`:

```
val lista = mutableListOf("Manzana", "Naranja", "Plátano", "Mango")
lista.add("Fresa")
```

Para insertar un elemento en una posición específica de la lista, se puede utilizar el método `add()` con un índice especificado:

```
val lista = mutableListOf("Manzana", "Naranja", "Plátano", "Mango")
lista.add(2, "Pera")
```

Para eliminar un elemento de la lista, se puede utilizar el método `remove()` o `removeAt()`, pasando el elemento o el índice del elemento a eliminar, respectivamente:

```
val lista = mutableListOf("Manzana", "Naranja", "Plátano", "Mango")
lista.remove("Naranja")
lista.removeAt(2)
```

Para eliminar todos los elementos de una lista mutable, se puede utilizar el método `clear()`:

```
val lista = mutableListOf("Manzana", "Naranja", "Plátano", "Mango")
lista.clear()
```

▼ Recorriendo una lista

En Kotlin, se puede recorrer una lista utilizando varios enfoques. Aquí hay algunos ejemplos:

- Recorrido con un ciclo for: Puedes usar un ciclo for para recorrer cada elemento de la lista. Por ejemplo:

```
val lista = listOf("hola", "mundo", "en", "Kotlin")

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

- Recorrido con un ciclo while: También puedes usar un ciclo while para recorrer cada elemento de la lista. Por ejemplo:

```
val lista = listOf("hola", "mundo", "en", "Kotlin")

var indice = 0
while (indice < lista.size) {
    println(lista[indice])
    indice++
}
```

- Recorrido con un iterador: Puedes obtener un iterador de la lista y recorrer cada elemento utilizando el método `next()` del iterador. Por ejemplo:

```
val lista = listOf("hola", "mundo", "en", "Kotlin")

val iterador = lista.iterator()
while (iterador.hasNext()) {
    val elemento = iterador.next()
    println(elemento)
}
```

- Recorrido con una función forEach: Puedes utilizar la función `forEach()` que se encuentra disponible en la clase List para recorrer la lista. Por ejemplo:

```
val lista = listOf("hola", "mundo", "en", "Kotlin")

lista.forEach {
    println(it)
}
```

▼ Funciones útiles para trabajar con listas en Kotlin

- `size`: Devuelve el número de elementos en la lista.

```
val lista = listOf("hola", "mundo", "en", "Kotlin")
val size = lista.size // devuelve 4
```

- `get`: Devuelve el elemento en la posición especificada.

```
val lista = listOf("hola", "mundo", "en", "Kotlin")
val elemento = lista.get(1) // devuelve "mundo"
```

- `indexOf`: Devuelve el índice de la primera aparición del elemento especificado, o -1 si el elemento no se encuentra en la lista.

```
val lista = listOf("hola", "mundo", "en", "Kotlin")
val indice = lista.indexOf("en") // devuelve 2
```

- `lastIndexOf`: Devuelve el índice de la última aparición del elemento especificado, o -1 si el elemento no se encuentra en la lista.

```
val lista = listOf("hola", "mundo", "en", "Kotlin", "en")
val indice = lista.lastIndexOf("en") // devuelve 4
```

- `contains`: Devuelve `true` si la lista contiene el elemento especificado, o `false` en caso contrario.

```
val lista = listOf("hola", "mundo", "en", "Kotlin")
val contieneMundo = lista.contains("mundo") // devuelve true
val contieneAdios = lista.contains("adios") // devuelve false
```

- `filter`: Devuelve una nueva lista que contiene solo los elementos que cumplen con la condición especificada.

```
val lista = listOf("hola", "mundo", "en", "Kotlin")
val filtrados = lista.filter { it.length > 3 } // devuelve ["mundo", "Kotlin"]
```

- `map`: Devuelve una nueva lista que contiene los resultados de aplicar la transformación especificada a cada elemento de la lista.

```
val lista = listOf("hola", "mundo", "en", "Kotlin")
val mapeados = lista.map { it.toUpperCase() } // devuelve ["HOLA", "MUNDO", "EN", "KOTLIN"]
```

- `distinct`: Devuelve una nueva lista que contiene solo los elementos únicos de la lista original.

```
val lista = listOf("hola", "mundo", "en", "Kotlin", "mundo")
val unicos = lista.distinct() // devuelve ["hola", "mundo", "en", "Kotlin"]
```

- `sorted`: Devuelve una nueva lista que contiene los elementos de la lista original ordenados en orden ascendente.

```
val lista = listOf(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)
val ordenados = lista.sorted() // devuelve [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

▼ Conjuntos en Kotlin

▼ ¿Qué es un conjunto?

En Kotlin, un conjunto es una colección de elementos únicos y no ordenados. En otras palabras, es una estructura de datos que almacena elementos en una lista que no permite elementos duplicados.

Los conjuntos en Kotlin proporcionan operaciones comunes como agregar o eliminar elementos, comprobar si un elemento está presente en el conjunto, iterar sobre los elementos del conjunto, etc. Además, también se pueden realizar operaciones matemáticas de conjuntos como unión, intersección y diferencia.

▼ Creación de conjuntos en Kotlin

En Kotlin, puedes crear conjuntos utilizando la función `setOf()` y especificando los elementos que deseas incluir en el conjunto. También puedes utilizar la función `mutableSetOf()` si necesitas crear un conjunto mutable que puedas agregar o eliminar elementos posteriormente.

Aquí hay algunos ejemplos de cómo crear conjuntos en Kotlin:

```
// Crear un conjunto inmutable de números
val conjuntoNumeros = setOf(1, 2, 3, 4, 5)

// Crear un conjunto mutable de nombres
val conjuntoNombres = mutableSetOf("Juan", "Ana", "Pedro", "Sofía")

// Crear un conjunto inmutable vacío
val conjuntoVacio = setOf<String>()
```

En el primer ejemplo, creamos un conjunto inmutable de números utilizando la función `setOf()` y especificando los elementos que deseamos incluir. El resultado es un conjunto de cinco números: 1, 2, 3, 4 y 5.

En el segundo ejemplo, creamos un conjunto mutable de nombres utilizando la función `mutableSetOf()` y especificando los nombres que deseamos incluir. Este conjunto puede ser modificado posteriormente utilizando las funciones de agregado o eliminación de elementos.

Finalmente, en el tercer ejemplo, creamos un conjunto inmutable vacío utilizando la función `setOf<String>()`. Este conjunto puede ser utilizado posteriormente para agregar elementos utilizando la función `plus()` o para crear un conjunto mutable utilizando la función `toMutableSet()`.

▼ Accediendo a los elementos de un conjunto

En Kotlin, puedes acceder a los elementos de un conjunto utilizando diferentes métodos proporcionados por la interfaz `Set`. Como los conjuntos no están ordenados, no puedes acceder a los elementos utilizando un índice, sino que debes utilizar otro enfoque. Por ejemplo:

- `contains()`: Comprueba si un elemento está presente en el conjunto. Devuelve `true` si el elemento está presente, `false` en caso contrario.

```
val conjunto = setOf("a", "b", "c")

if (conjunto.contains("b")) {
    println("El conjunto contiene el elemento 'b'")
}
```

- `forEach()`: Itera sobre todos los elementos del conjunto y realiza una acción en cada elemento.

```
val conjunto = setOf("a", "b", "c")

conjunto.forEach { elemento ->
    println("Elemento: $elemento")
}
```

- `iterator()`: Devuelve un iterador que puedes utilizar para iterar sobre los elementos del conjunto.

```
val conjunto = setOf("a", "b", "c")

val iterador = conjunto.iterator()

while (iterador.hasNext()) {
    val elemento = iterador.next()
    println("Elemento: $elemento")
}
```

- `elementAt()`: Devuelve el elemento en una posición dada. Debido a que los conjuntos no están ordenados, este método no es muy útil para los conjuntos.

```
val conjunto = setOf("a", "b", "c")

val elemento = conjunto.elementAt(1)

println("El segundo elemento del conjunto es '$elemento'")
```

```
fun main(){
    var lista1 = mutableListOf("Manzana", "Naranja", "Plátano", "Mango")

    //accediendo a ella
    val segundoElemento = lista1[1]
    val tercerElemento = lista1.get(2)

    println("El elemento 1 y 2 es: $segundoElemento , $tercerElemento")
}fun main(){
    var lista1 = mutableListOf("Manzana", "Naranja", "Plátano", "Mango")

    //accediendo a ella
    val segundoElemento = lista1[1]
    val tercerElemento = lista1.get(2)

    println("El elemento 1 y 2 es: $segundoElemento , $tercerElemento")
}
```

En general, los conjuntos en Kotlin se utilizan principalmente para comprobar la presencia de elementos y realizar operaciones de conjuntos como unión, intersección y diferencia. Si necesitas acceder a los elementos de una colección utilizando un índice, deberías considerar el uso de una lista en su lugar.

▼ Modificando los elementos de un conjunto

En Kotlin, los conjuntos inmutables no permiten la modificación de sus elementos. Si necesitas modificar los elementos de un conjunto, debes utilizar un conjunto mutable.

Aquí hay algunos ejemplos de cómo modificar los elementos de un conjunto mutable en Kotlin:

- Agregar elementos: Puedes agregar elementos a un conjunto mutable utilizando la función `add()` o la operación `+=`.

```
val conjuntoMutable = mutableSetOf("a", "b", "c")

conjuntoMutable.add("d")
conjuntoMutable += "e"

println(conjuntoMutable) // Imprime [a, b, c, d, e]
```

- Eliminar elementos: Puedes eliminar elementos de un conjunto mutable utilizando la función `remove()` o la operación `-=`.

```
val conjuntoMutable = mutableSetOf("a", "b", "c")

conjuntoMutable.remove("b")
conjuntoMutable -= "c"

println(conjuntoMutable) // Imprime [a]
```

- Actualizar elementos: Puedes actualizar un elemento de un conjunto mutable eliminando el elemento antiguo y agregando el nuevo elemento.

```
val conjuntoMutable = mutableSetOf("a", "b", "c")

conjuntoMutable.remove("b")
conjuntoMutable += "d"

println(conjuntoMutable) // Imprime [a, c, d]
```

En general, las operaciones de modificación en los conjuntos mutables en Kotlin son fáciles de realizar y se asemejan a las operaciones realizadas en las listas mutables. Ten en cuenta que las operaciones de modificación pueden cambiar el orden de los elementos en el conjunto mutable ya que los conjuntos no están ordenados.

▼ Recorriendo un conjunto

En Kotlin, puedes recorrer un conjunto utilizando diferentes métodos proporcionados por la interfaz `Set`.

Aquí hay algunos métodos que puedes utilizar para recorrer un conjunto en Kotlin:

- `forEach()`: Itera sobre todos los elementos del conjunto y realiza una acción en cada elemento.

```
val conjunto = setOf("a", "b", "c")

conjunto.forEach { elemento ->
    println("Elemento: $elemento")
}
```

- `iterator()`: Devuelve un iterador que puedes utilizar para iterar sobre los elementos del conjunto.

```
val conjunto = setOf("a", "b", "c")

val iterador = conjunto.iterator()

while (iterador.hasNext()) {
    val elemento = iterador.next()
    println("Elemento: $elemento")
}
```

- `for-in loop`: Puedes utilizar un bucle `for-in` para recorrer los elementos del conjunto.

```
val conjunto = setOf("a", "b", "c")

for (elemento in conjunto) {
    println("Elemento: $elemento")
}
```

En general, los conjuntos en Kotlin se utilizan principalmente para comprobar la presencia de elementos y realizar operaciones de conjuntos como unión, intersección y diferencia. Si necesitas acceder a los elementos de una colección utilizando un índice, deberías considerar el uso de una lista en su lugar.

▼ Funciones útiles para trabajar con conjuntos en Kotlin

- `add(elemento: E): Boolean`: Agrega el elemento especificado al conjunto y devuelve `true` si el conjunto se modificó como resultado de la operación.

```
val conjunto = mutableSetOf("a", "b", "c")
conjunto.add("d")
```

- `remove(elemento: E): Boolean`: Elimina el elemento especificado del conjunto y devuelve `true` si el conjunto se modificó como resultado de la operación.

```
val conjunto = mutableSetOf("a", "b", "c")
conjunto.remove("b")
```

- `contains(elemento: E): Boolean`: Devuelve `true` si el conjunto contiene el elemento especificado.

```
val conjunto = setOf("a", "b", "c")
val contieneA = conjunto.contains("a") // true
val contieneD = conjunto.contains("d") // false
```

- `isEmpty(): Boolean`: Devuelve `true` si el conjunto está vacío.

```
val conjunto = setOf("a", "b", "c")
val estaVacio = conjunto.isEmpty() // false
```

- `size: Int`: Devuelve la cantidad de elementos en el conjunto.

```
val conjunto = setOf("a", "b", "c")
val cantidadElementos = conjunto.size // 3
```

- `union(otroConjunto: Set<E>): Set<E>`: Devuelve un nuevo conjunto que contiene todos los elementos del conjunto actual y del otro conjunto especificado.

```
val conjunto1 = setOf("a", "b", "c")
val conjunto2 = setOf("c", "d", "e")
val union = conjunto1.union(conjunto2) // [a, b, c, d, e]
```

- `intersect(otroConjunto: Set<E>): Set<E>`: Devuelve un nuevo conjunto que contiene los elementos que están presentes tanto en el conjunto actual como en el otro conjunto especificado.

```
val conjunto1 = setOf("a", "b", "c")
val conjunto2 = setOf("c", "d", "e")
val interseccion = conjunto1.intersect(conjunto2) // [c]
```

- `subtract(otroConjunto: Set<E>): Set<E>`: Devuelve un nuevo conjunto que contiene los elementos del conjunto actual que no están presentes en el otro conjunto especificado.

```
val conjunto1 = setOf("a", "b", "c")
val conjunto2 = setOf("c", "d", "e")
val diferencia = conjunto1.subtract(conjunto2) // [a, b]
```

▼ Mapas en Kotlin

▼ ¿Qué es un mapa?

En Kotlin, un mapa es una colección que almacena pares de elementos clave-valor. Cada elemento del mapa se identifica por su clave única y se puede acceder a su valor correspondiente utilizando esta clave.

En términos generales, un mapa es similar a un diccionario en otros lenguajes de programación. Cada clave se utiliza para buscar el valor correspondiente, lo que permite una búsqueda muy rápida y eficiente.

▼ Creación de mapas en Kotlin

Los mapas se pueden crear utilizando la interfaz `Map`. En Kotlin, hay dos tipos de mapas:

1. `mutableMapOf()`: Crea un mapa mutable que se puede modificar.

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

2. `mapOf()`: Crea un mapa inmutable que no se puede modificar.

```
val mapa2 = mapOf(
    1 to "uno",
    2 to "dos"
)
```

3. Crear un mapa vacío:

```
val mapa = emptyMap<String, Int>()
```

En los ejemplos anteriores, las claves son `String` o `Int` y los valores son `Int` o `String`. Sin embargo, los mapas en Kotlin pueden contener claves y valores de cualquier tipo. Por ejemplo, un mapa podría tener claves de tipo `Boolean` y valores de tipo `Any`, o claves de tipo `CharSequence` y valores de tipo `Double`.

▼ Accediendo a los elementos de un mapa

En Kotlin, se puede acceder a los elementos de un mapa utilizando las claves. Para acceder al valor asociado con una clave en particular, se puede utilizar la función `get()`, o la sintaxis del operador de índice. Por ejemplo:

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

val valor = mapa.get("a")
val valor2 = mapa["a"]
```

En este ejemplo, la variable `valor` contendría el valor `1`, ya que `"a"` es la clave del primer elemento del mapa. La variable `valor2` también contendría el valor `1`, ya que se utiliza la sintaxis del operador de índice para acceder al valor asociado con la clave `"a"`.

Si la clave no existe en el mapa, la función `get()` devuelve `null`. También se puede utilizar la función `getOrDefault()` para obtener el valor asociado con una clave, o un valor predeterminado si la clave no existe en el mapa:

```
val valor3 = mapa.getOrDefault("d", 0)
```

En este ejemplo, `valor3` contendría el valor `0`, ya que la clave `"d"` no existe en el mapa.

también se puede acceder a todas las claves o todos los valores en un mapa utilizando las funciones `keys` y `values`, respectivamente:

```
val claves = mapa.keys
val valores = mapa.values
```

En este ejemplo, la variable `claves` contendría una lista de todas las claves en el mapa (`["a", "b", "c"]`), y la variable `valores` contendría una lista de todos los valores en el mapa (`[1, 2, 3]`).

▼ Modificando los elementos de un mapa

En Kotlin, los mapas mutables se pueden modificar utilizando las funciones `put()` y `putAll()`. La función `put()` se utiliza para agregar o actualizar un elemento en el mapa, mientras que la función `putAll()` se utiliza para agregar múltiples elementos a la vez. Por ejemplo:

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

mapa.put("d", 4)
mapa.put("b", 5)
```

En este ejemplo, se agrega un nuevo elemento al mapa con la clave `"d"` y el valor `4`, y se actualiza el valor del elemento existente con la clave `"b"` a `5`.

También se puede utilizar la sintaxis del operador de índice para agregar o actualizar elementos en el mapa:

```
mapa["e"] = 6
mapa["a"] = 7
```

En este caso, se agrega un nuevo elemento al mapa con la clave `"e"` y el valor `6`, y se actualiza el valor del elemento existente con la clave `"a"` a `7`.

Para eliminar un elemento de un mapa mutable, se puede utilizar la función `remove()`. Por ejemplo:

```
mapa.remove("b")
```

En este ejemplo, se elimina el elemento del mapa con la clave `"b"`.

También se puede eliminar todos los elementos de un mapa utilizando la función `clear()`:

```
mapa.clear()
```

En este ejemplo, se eliminan todos los elementos del mapa, dejándolo vacío.

Tenga en cuenta que no es posible modificar los elementos de un mapa inmutable en Kotlin. Si se desea cambiar los elementos de un mapa inmutable, se debe crear un nuevo mapa con los elementos modificados.

▼ Recorriendo un mapa

Para recorrer un mapa en Kotlin, se puede utilizar un bucle `for` con la función `forEach()` del mapa. Por ejemplo:

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

mapa.forEach { clave, valor ->
    println("$clave -> $valor")
}
```

En este ejemplo, se recorre el mapa utilizando la función `forEach()`, que toma una función lambda con dos parámetros: la clave y el valor de cada elemento del mapa. Dentro de la función lambda, se puede hacer lo que se desee con la clave y el valor. En este caso, se imprime cada clave y valor en la consola.

También se puede utilizar un bucle `for` con las propiedades `keys` o `values` de un mapa para recorrer todas las claves o todos los valores del mapa, respectivamente. Por ejemplo:

```
for (clave in mapa.keys) {
    val valor = mapa.get(clave)
    println("$clave -> $valor")
}

for (valor in mapa.values) {
    println("$valor")
}
```

En este ejemplo, se utilizan los bucles `for` para recorrer todas las claves y valores del mapa. En el primer bucle, se obtiene el valor asociado con cada clave utilizando la función `get()`. En el segundo bucle, se imprime cada valor directamente.

También se puede utilizar la sintaxis del operador de índice para acceder a los elementos del mapa, y luego utilizar un bucle `for` para recorrer todas las claves o valores del mapa. Por ejemplo:

```
for (clave in mapa.keys) {
    val valor = mapa[clave]
    println("$clave -> $valor")
}

for (valor in mapa.values) {
    println("$valor")
}
```

En este caso, se utiliza la sintaxis del operador de índice para acceder a los valores del mapa.

▼ Funciones útiles para trabajar con mapas en Kotlin

- `get()`: Devuelve el valor asociado con la clave dada, o `null` si la clave no está presente en el mapa.

```
fun main() {
    val mapa = mapOf("a" to 1, "b" to 2, "c" to 3)
    val valorA = mapa.get("a")
    val valorD = mapa.get("d")

    println("Valor de 'a': $valorA") // Imprime "Valor de 'a': 1"
    println("Valor de 'd': $valorD") // Imprime "Valor de 'd': null"
}
```

- `getOrElse()`: Devuelve el valor asociado con la clave dada, o el resultado de la función lambda si la clave no está presente en el mapa.

```
fun main() {
    val mapa = mapOf("a" to 1, "b" to 2, "c" to 3)
    val valorA = mapa.getOrElse("a") { 0 }
    val valorD = mapa.getOrElse("d") { 0 }

    println("Valor de 'a': $valorA") // Imprime "Valor de 'a': 1"
    println("Valor de 'd': $valorD") // Imprime "Valor de 'd': 0"
}
```

- `getOrDefault()`: Devuelve el valor asociado con la clave dada, o el valor predeterminado dado si la clave no está presente en el mapa.

```
fun main() {
    val mapa = mapOf("a" to 1, "b" to 2, "c" to 3)
    val valorA = mapa.getOrDefault("a", 0)
    val valorD = mapa.getOrDefault("d", 0)

    println("Valor de 'a': $valorA") // Imprime "Valor de 'a': 1"
    println("Valor de 'd': $valorD") // Imprime "Valor de 'd': 0"
}
```

- `containsKey()`: Devuelve `true` si el mapa contiene la clave dada, o `false` de lo contrario.

```
fun main() {
    val mapa = mapOf("a" to 1, "b" to 2, "c" to 3)
```

```
    if (mapa.containsKey("a")) {
        println("El mapa contiene la clave 'a'")
    } else {
        println("El mapa no contiene la clave 'a'")
    }

    if (mapa.containsKey("d")) {
        println("El mapa contiene la clave 'd'")
    } else {
        println("El mapa no contiene la clave 'd'")
    }
}
```

- `containsValue()`: Devuelve `true` si el mapa contiene el valor dado, o `false` de lo contrario.

```
fun main() {
    val mapa = mapOf("a" to 1, "b" to 2, "c" to 3)

    if (mapa.containsValue(1)) {
        println("El mapa contiene el valor 1")
    } else {
        println("El mapa no contiene el valor 1")
    }

    if (mapa.containsValue(4)) {
        println("El mapa contiene el valor 4")
    } else {
        println("El mapa no contiene el valor 4")
    }
}
```

- `keys`: Devuelve una colección de todas las claves del mapa.

```
fun main() {
    val mapa = mapOf("a" to 1, "b" to 2, "c" to 3)

    val claves = mapa.keys

    println("Claves del mapa: $claves")
}
```

- `values`: Devuelve una colección de todos los valores del mapa.

```
fun main() {
    val mapa = mapOf("a" to 1, "b" to 2, "c" to 3)

    val valores = mapa.values

    println("Valores del mapa: $valores")
}
```

- `size`: Devuelve el número de elementos en el mapa.

```
fun main() {
    val mapa = mapOf("a" to 1, "b" to 2, "c" to 3)

    val tamano = mapa.size

    println("El tamaño del mapa es $tamano")
}
```

- `isEmpty()`: Devuelve `true` si el mapa está vacío, o `false` de lo contrario.

```
```

- `put()`: Agrega o actualiza un elemento en el mapa con la clave y el valor dados.

```
fun main() {
    val mapa1 = mapOf<String, Int>()
    val mapa2 = mapOf("a" to 1, "b" to 2, "c" to 3)

    if (mapa1.isEmpty()) {
        println("El mapa1 está vacío")
    } else {
        println("El mapa1 no está vacío")
    }

    if (mapa2.isEmpty()) {
        println("El mapa2 está vacío")
    } else {
        println("El mapa2 no está vacío")
    }
}
```

- `putAll()`: Agrega múltiples elementos al mapa a la vez.

```
fun main() {
    val mapa1 = mapOf("a" to 1, "b" to 2)
    val mapa2 = mutableMapOf<String, Int>("c" to 3, "d" to 4)

    println("Mapa 1 antes de putAll(): $mapa1")
    println("Mapa 2 antes de putAll(): $mapa2")

    mapa2.putAll(mapa1)

    println("Mapa 1 después de putAll(): $mapa1")
    println("Mapa 2 después de putAll(): $mapa2")
}
```

- `remove()`: Elimina el elemento del mapa con la clave dada.

```
fun main() {
    val mapa = mutableMapOf("a" to 1, "b" to 2, "c" to 3)

    println("Mapa antes de eliminar: $mapa")

    mapa.remove("b")

    println("Mapa después de eliminar: $mapa")
}
```

- `clear()`: Elimina todos los elementos del mapa.

```
fun main() {
    val mapa = mutableMapOf("a" to 1, "b" to 2, "c" to 3)

    println("Mapa antes de clear: $mapa")

    mapa.clear()

    println("Mapa después de clear: $mapa")
}
```

▼ Pares en Kotlin

▼ ¿Qué es un par?

Un par en Kotlin es una estructura de datos que contiene exactamente dos elementos. Se puede utilizar para devolver dos valores desde una función, para almacenar dos valores relacionados juntos o para cualquier otra situación en la que necesite trabajar con dos valores juntos.

▼ Creación de pares en Kotlin

En Kotlin, un par se crea utilizando la clase `Pair`, que tiene dos argumentos de tipo genérico `A` y `B` que representan los dos elementos del par.

Aquí hay algunos ejemplos de cómo crear un par en Kotlin:

```
// Crear un par de una cadena y un número entero
val par1 = Pair("hola", 42)

// Crear un par de dos números flotantes
val par2 = Pair(3.14f, 2.718f)

// Crear un par de un booleano y un carácter
val par3 = Pair(true, 'x')
```

En estos ejemplos, creamos tres pares utilizando la clase `Pair`. En el primer ejemplo, el primer elemento es una cadena y el segundo es un número entero. En el segundo ejemplo, ambos elementos son números flotantes. En el tercer ejemplo, el primer elemento es un booleano y el segundo es un carácter.

También se puede utilizar la función de extensión `to()`
para crear pares de manera más concisa, como se muestra en el siguiente ejemplo:

```
// Crear un par de una cadena y un número entero
val par4 = "hola" to 42

// Crear un par de dos números flotantes
val par5 = 3.14f to 2.718f

// Crear un par de un booleano y un carácter
val par6 = true to 'x'
```

En este ejemplo, creamos los mismos tres pares utilizando la función de extensión `to()` en lugar de la clase `Pair`. La función `to()` toma el primer elemento como su receptor (es decir, la cadena o el número flotante) y el segundo elemento como su argumento.

▼ Accediendo a los elementos de un par

Para acceder a los elementos de un par en Kotlin, podemos utilizar las propiedades `first` y `second`, que corresponden al primer y segundo elemento del par, respectivamente.

Aquí hay un ejemplo de cómo acceder a los elementos de un par en Kotlin:

```
// Crear un par de una cadena y un número entero
val par = Pair("hola", 42)

// Acceder al primer y segundo elemento del par
val primerElemento = par.first  // "hola"
val segundoElemento = par.second  // 42
```

En este ejemplo, creamos un par de una cadena y un número entero, y luego utilizamos las propiedades `first` y `second` para acceder a los elementos del par y guardarlos en variables separadas.

También se puede utilizar la desestructuración para asignar los elementos del par a variables individuales, como se muestra en el siguiente ejemplo:

```
// Crear un par de una cadena y un número entero
val par = Pair("hola", 42)

// Desestructurar el par para asignar los elementos a variables individuales
val (primerElemento, segundoElemento) = par
```

En este ejemplo, utilizamos la desestructuración para asignar los elementos del par a las variables `primerElemento` y `segundoElemento`. La sintaxis de la desestructuración es similar a la de la declaración de variables, pero en lugar de asignar un valor a una sola variable, se asignan los valores a varias variables a la vez.

▼ Modificando los elementos de un par

En Kotlin, los objetos `Pair` son inmutables, lo que significa que no se pueden modificar una vez creados. Por lo tanto, no se pueden modificar los elementos de un par una vez que se han establecido.

Si necesitas tener una estructura de datos que permita la modificación de sus elementos, puedes considerar el uso de otras estructuras de datos, como una lista o un mapa, que sí son mutables y permiten agregar, eliminar y actualizar elementos.

▼ Recorriendo un par

En Kotlin, un par (también conocido como tupla) es un objeto que contiene dos elementos. Por lo tanto, para recorrer un par, podemos utilizar cualquiera de los métodos disponibles para recorrer una estructura de datos con dos elementos.

Por ejemplo, podemos utilizar la desestructuración para acceder a los elementos del par y luego realizar cualquier operación que deseemos. Aquí hay un ejemplo:

```
// Crear un par de una cadena y un número entero
val par = Pair("hola", 42)

// Utilizar la desestructuración para acceder a los elementos del par
val (primerElemento, segundoElemento) = par

// Imprimir los elementos del par
println(primerElemento) // imprime "hola"
println(segundoElemento) // imprime 42
```

En este ejemplo, utilizamos la desestructuración para acceder a los elementos del par y luego los imprimimos en la consola.

También podemos recorrer un par utilizando una función de orden superior, como `forEach` o `map`. Aquí hay un ejemplo:

```

```

En este ejemplo, utilizamos la función `forEach`
para recorrer el par y luego imprimimos cada elemento en la consola. La variable `elemento` contiene cada elemento del par en cada iteración.

Ten en cuenta que, dado que un par solo tiene dos elementos, es posible que no tenga sentido recorrerlo de la misma manera que recorremos una lista o un mapa. En general, utilizamos pares para empaquetar dos valores relacionados, por lo que acceder a ellos a través de la desestructuración o simplemente accediendo a `first` y `second` suele ser suficiente.

▼ Funciones útiles para trabajar con pares en Kotlin

Kotlin proporciona varias funciones útiles para trabajar con pares. Aquí hay algunas de las más comunes:

- `first` y `second`: Estas son propiedades de solo lectura que nos permiten acceder al primer y segundo elemento de un par. Por ejemplo:

```
val par = Pair("hola", 42)
val primerElemento = par.first // "hola"
val segundoElemento = par.second // 42
```

- `toString()`: Devuelve una cadena que representa el par. Por ejemplo:

```
val par = Pair("hola", 42)
val cadena = par.toString() // "(hola, 42)"
```

- `copy()`: Devuelve una copia del par con los mismos elementos. Podemos proporcionar nuevos valores para el primer y segundo elemento como argumentos. Por ejemplo:

```
val par1 = Pair("hola", 42)
val par2 = par1.copy() // crea una copia de par1, es decir (hola, 42)
val par3 = par1.copy("adiós", 99) // crea una copia de par1 con el primer elemento "adiós" y el segundo elemento 99
```

- `equals()`: Compara si dos pares son iguales. Dos pares son iguales si y solo si sus primeros y segundos elementos son iguales. Por ejemplo:

```
val par1 = Pair("hola", 42)
val par2 = Pair("hola", 42)
val par3 = Pair("adiós", 99)
val sonIguales = par1 == par2 // true
val noSonIguales = par1 == par3 // false
```

- `hashCode()`: Devuelve el código hash del par. Este método es útil cuando queremos almacenar pares en estructuras de datos como conjuntos o mapas.

▼ Prácticas de estructuras de datos en Kotlin

▼ Ejercicios prácticos para aplicar los conceptos aprendidos

▼ Arreglos Kotlin

▼ Ejercicio 1

Cree un programa que cree un arreglo de enteros que contenga los números del 1 al 20. Luego, imprima los números del arreglo en orden ascendente.

▼ Ejercicio 2

Cree un programa que cree un arreglo de Strings que contenga los nombres de sus mejores amigos. Luego, imprima los nombres del arreglo en orden alfabético.

- Listas en Kotlin
  - Ejercicio 1

    Cree un programa que cree una lista de enteros que contenga los números del 1 al 20. Luego, calcule el promedio de los números de la lista y almacene el resultado en una variable.

  - Ejercicio 2

    Cree un programa que cree una lista de enteros que contenga los números del 1 al 5. Luego, multiplique los números de la lista y almacene el resultado en una variable.

- Conjuntos en Kotlin
  - Ejercicio 1

    Crea dos conjuntos de enteros y realiza las siguientes operaciones:

    - Unión de los conjuntos
    - Intersección de los conjuntos
    - Diferencia de los conjuntos

  - Ejercicio 2

    Crea un conjunto de Strings con los nombres de tus amigos y pregunta al usuario si un nombre dado se encuentra en el conjunto. Si el nombre está en el conjunto, imprime un mensaje indicando que el amigo está en el conjunto. Si el nombre no está en el conjunto, agrega el nombre al conjunto y luego imprime un mensaje indicando que se ha agregado un nuevo amigo.

- Mapas en Kotlin
  - Ejercicio 1

    Crea un mapa que asocie nombres de personas con sus edades y luego imprime las edades de las personas cuyos nombres comienzan con la letra "A".

  - Ejercicio 2

    Crea un mapa que asocie nombres de ciudades con sus poblaciones y luego pregunta al usuario por el nombre de una ciudad y muestra su población. Si la ciudad no está en el mapa, muestra un mensaje indicando que la ciudad no está registrada.

- Pares en Kotlin
  - Ejercicio 1

    Crea un programa que cree una lista de pares que contengan el nombre y la edad de varias personas. Luego, itera sobre la lista y imprime el nombre y la edad de cada persona.

  - Ejercicio 2

    Crea una lista de pares enteros y encuentra la multiplicación de todos los elementos en la lista.

- Solución a los ejercicios prácticos
  - Solución de arreglos
    - Ejercicio 1

```
fun main() {
    // Crear un arreglo de enteros del 1 al 20
    val numeros = IntArray(20) { it + 1 }

    // Imprimir los números del arreglo en orden ascendente
    numeros.forEach { numero ->
        println(numero)
```

```
    }
}
```

se utiliza la función `sorted()` para ordenar los nombres en orden alfabético. Esta función devuelve una nueva lista con los elementos del arreglo ordenados

- Ejercicio 2

```
fun main() {
    // Crear un arreglo de Strings
    val amigos = arrayOf("Ana", "Carlos", "Diego", "Fernanda", "Luisa")

    // Ordenar los nombres en orden alfabético
    val nombresOrdenados = amigos.sorted()

    // Imprimir los nombres ordenados
    nombresOrdenados.forEach { nombre ->
        println(nombre)
    }
}
```

▼ Solución de listas

- Ejercicio 1

```
fun main() {
    // Crear una lista de enteros con los números del 1 al 20
    val listaNumeros = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)

    // Calcular el promedio de los números de la lista
    val promedio = listaNumeros.average()

    // Almacenar el resultado en una variable
    val resultado = promedio
    println("El promedio de los números es: $resultado")
}
```

- Ejercicio 2

```
fun main(){
    // Crear una lista de enteros con los números del 1 al 5
    val lista = listOf(1,2,3,4,5)

    //Multiplica los números de la lista y almacena el resultado en una variable
    var resultado = 1
    lista.forEach { resultado *= it }

    //imprimir el resultado final
    println(resultado)
}
```

▼ Solución de Conjuntos

- Ejercicio 1

```
fun main(){
    //creación del conjunto con numeros enteros
    val conjuntoA = setOf(1, 2, 3, 4, 5)
    val conjuntoB = setOf(4, 5, 6, 7, 8)

    //union de los conjuntos
```

```
    val union = conjuntoA.union(conjuntoB)
    println("Unión: $union")

    //intersección de los conjuntos
    val interseccion = conjuntoA.intersect(conjuntoB)
    println("Intersección: $interseccion")

    //Diferencia de los conjuntos
    val diferencia = conjuntoA.subtract(conjuntoB)
    println("Diferencia: $diferencia")
}
```

- Ejercicio 2

```
fun main(){
    //creando conjunto de Strings
    val amigos = mutableSetOf("Ana", "Juan", "María", "Pedro")

    //Ingreso por el teclado
    println("Escribe el nombre de un amigo:")
    val nombre = readLine()!!

    if (amigos.contains(nombre)) {
        println("El amigo $nombre está en el conjunto")
    } else {
        amigos.add(nombre)
        println("Se ha agregado el amigo $nombre al conjunto")
    }
}
```

▼ Solución de Mapas

- Ejercicio 1

```
fun main(){
    //creacion de mapa con nombres y edades
    val edades = mapOf("Ana" to 25, "Carlos" to 32, "Alicia" to 28, "Pedro" to 29)

    //funcion para elegir las personas que empiecen por el nombre a
    val edadesA = edades.filterKeys { it.startsWith("A") }.values

    println("Edades de las personas cuyo nombre comienza con A: $edadesA")
}
```

- Ejercicio 2

```
fun main(){
    val poblaciones = mapOf("Madrid" to 3207247, "Barcelona" to 1620343, "Valencia" to 791413, "Sevilla" to 688711)

    println("Escribe el nombre de una ciudad:")
    val ciudad = readLine()!!

    val poblacion = poblaciones[ciudad]

    if (poblacion != null) {
        println("La población de $ciudad es $poblacion")
    } else {
        println("La ciudad $ciudad no está registrada")
    }
}
```

▼ Solución de Pares

- Ejercicio 1

```
fun main() {
    val lista = listOf(
        Pair("Juan", 25),
        Pair("Maria", 30),
        Pair("Pedro", 28)
    )
    lista.forEach { persona ->
        val nombre = persona.first
        val edad = persona.second
        println("$nombre tiene $edad años")
    }
}
```

- Ejercicio 2

```
fun main(){
    val listadepares = listOf(Pair(2, 3), Pair(5, 4), Pair(1, 6))

    var resultado = 1
    listadepares.forEach { resultado *= it.first * it.second }

    println(resultado)
}
```