Funciones de orden superior

Una función de orden superior (FOS) es una función que cumple una de las siguientes condiciones:

- recibe una función como parámetro
- devuelve una función

Las FOS están contempladas en Kotlin y permiten tratar a las funciones como cualquier otro valor.

```
fun <T, R> Collection<T>.fold(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

Funciones como parámetros

Funciones con parámetros de entrada

Ahora vamos a ver las funciones con parámetros de entrada, que son iguales, pero al llamarlas habrá que mandarle las variables que necesite.

Funciones con parámetros de salida

Nos queda por ver como una función puede devolver un resultado o lo que haga nuestro método. La única limitación es que solo se puede devolver un parámetro, aunque para eso tenemos los métodos.

añadimos los parámetros de entrada, pero esta vez, al cerrar los paréntesis pondremos el tipo de variable que debe devolver nuestra función. Luego la función hará todo lo que tenga que hacer y cuando tenga el resultado, lo devolveremos con la palabra clave return.

Recursión o recursividad en Kotlin

Una función sí puede ser llamada dentro de sí misma, y a eso se le llama recursión o recursividad.

No solo Kotlin lo permite; lo permiten varios lenguajes. Recuerda que estas funciones deben tener una salida.

List

La lista <T> almacena elementos en un orden específico y les proporciona acceso indexado. Los índices comienzan desde cero, el índice del primer elemento, y van a lastIndex, que es (list.size - 1).

```
val numbers = listOf("one", "two", "three", "four")
println("Number of elements: ${numbers.size}")
println("Third element: ${numbers.get(2)}")
println("Fourth element: ${numbers[3]}")
println("Index of element \"two\" ${numbers.indexOf("two")}")
```

Los elementos de la list (incluidos los nulos) pueden duplicarse: una lista puede contener cualquier número de objetos iguales o ocurrencias de un solo objeto. Dos lists se consideran iguales si tienen los mismos tamaños y elementos estructuralmente iguales en las mismas posiciones.

```
val bob = Person("Bob", 31)
val people = listOf<Person>(Person("Adam", 20), bob, bob)
val people2 = listOf<Person>(Person("Adam", 20), Person("Bob", 31), bob)
println(people == people2)
bob.age = 32
println(people == people2)
```

MutableList es una lista con operaciones de escritura específicas de la lista, por ejemplo, para agregar o eliminar un elemento en una posición específica.

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
numbers.removeAt(1)
numbers[0] = 0
numbers.shuffle()
println(numbers)
```

Como puede ver, en algunos aspectos las listas son muy similares a las matrices. Sin embargo, hay una diferencia importante: el tamaño de una matriz se define en la inicialización y nunca cambia; a su vez, una lista no tiene un tamaño predefinido; El tamaño de una lista se puede cambiar como resultado de operaciones de escritura: agregar, actualizar o eliminar elementos.

En Kotlin, la implementación predeterminada de List es ArrayList, que se puede considerar como una matriz redimensionable.

Filter

El filtrado es una de las tareas más populares en el procesamiento de la colección. En Kotlin, las condiciones de filtrado están definidas por predicados: funciones lambda que toman un elemento de colección y devuelven un valor booleano: verdadero significa que el elemento dado coincide con el predicado, falso significa lo contrario.

La biblioteca estándar contiene un grupo de funciones de extensión que le permiten filtrar colecciones en una sola llamada. Estas funciones dejan la colección original sin cambios, por lo que están disponibles tanto para colecciones mutables como de solo lectura.

Para operar el resultado del filtrado, debe asignarlo a una variable o encadenar las funciones después del filtrado.

Filtrado por predicado

La función básica de filtrado es filter (). Cuando se llama con un predicado, filter () devuelve los elementos de la colección que coinciden. Tanto para Lista como para Conjunto, la colección resultante es una Lista, para Mapa también es un Mapa.

```
val numbers = listOf("one", "two", "three", "four")
val longerThan3 = numbers.filter { it.length > 3 }
println(longerThan3)

val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}
println(filteredMap)
```

Los predicados en filter () solo pueden verificar los valores de los elementos. Si desea usar posiciones de elementos en el filtro, use filterIndexed (). Toma un predicado con dos argumentos: el índice y el valor de un elemento.

Para filtrar colecciones por condiciones negativas, use filterNot (). Devuelve una lista de elementos para los cuales el predicado produce falso.

```
val numbers = listOf("one", "two", "three", "four")

val filteredIdx = numbers.filterIndexed { index, s -> (index != 0) && (s.length < 5)  }
val filteredNot = numbers.filterNot { it.length <= 3 }

println(filteredIdx)
println(filteredNot)
```

También hay funciones que limitan el tipo de elemento filtrando elementos de un tipo dado:

filterIsInstance () devuelve elementos de colección de un tipo dado. Al ser llamado en una List <Any>, filterIsInstance <T> () devuelve una List <T>, lo que le permite llamar a funciones del tipo T en sus elementos.

```
val numbers = listOf(null, 1, "two", 3.0, "four")
println("All String elements in upper case:")
numbers.filterIsInstance<String>().forEach {
    println(it.toUpperCase())
}
```

filterNotNull () devuelve todos los elementos no nulos. Al ser llamado en una List <T?>, FilterNotNull () devuelve una List <T: Any>, lo que le permite tratar los elementos como objetos no nulos.