TECNOLÓGICO
NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE NUEVO LAREDO

Materia: Android

Facilitador: Ing. Ahumada

Investigación 2

Alumno: Yesenia Mendoza Martínez.

Especialidad: Ingeniería en Sistemas Computacionales.

Semestre:8 ° #**Control:** 16100236

Nuevo Laredo Tamaulipas. Febrero 2020

Kotlin es un lenguaje orientado a objetos, pero introduce características existentes en los lenguajes funcionales que nos permiten crear un código más claro y expresivo.

Funciones de orden superior.

Una de las características del paradigma de la programación funcional son las funciones de orden superior.

Las funciones de orden superior son funciones que pueden recibir como parámetros otras funciones y/o devolverlas como resultados.

En el siguiente ejemplo se muestra un ejercicio donde se puede apreciar una función de orden superior:

```
fun operar(v1: Int, v2: Int, fn: (Int, Int) -> Int) : Int {
    return fn(v1, v2)
}

fun sumar(x1: Int, x2: Int) = x1 + x2

fun restar(x1: Int, x2: Int) = x1 - x2

fun multiplicar(x1: Int, x2: Int) = x1 * x2

fun dividir(x1: Int, x2: Int) = x1 / x2

fun main(parametro: Array<String>) {
    val resu1 = operar(10, 5, ::sumar)
    println("La suma de 10 y 5 es $resu1")
    val resu2 = operar(5, 2, ::sumar)
    println("La suma de 5 y 2 es $resu2")
    println("La resta de 100 y 40 es ${operar(100, 40, ::restar)}")
    println("El producto entre 5 y 20 es ${operar(5, 20, ::multiplicar)}")
    println("La división entre 10 y 5 es ${operar(10, 5, ::dividir)}")
}
```

De las 6 funciones del código anterior la única función de orden superior es la llamada "operar".

El tercer parámetro de esta función se llama "fn" y es de tipo función.
Cuando un parámetro es de tipo función debemos indicar los parámetros que tiene dicha función (en este caso tiene dos parámetros enteros) y luego del operador -> el tipo de dato que retorna esta función.

```
fn: (Int, Int) -> Int
```

Cuando tengamos una función como parámetro que no retorne dato se indica el tipo Unit, por ejemplo:

```
fn: (Int, Int) -> Unit
```

Funciones como parámetro.

Se puede hacer referencia a una función sin realmente llamarla prefijando el nombre de la función con `::` . Esto se puede pasar a una función que acepta alguna otra función como parámetro.

```
fun addTwo(x: Int) = x + 2
listOf(1, 2, 3, 4).map(::addTwo) # => [3, 4, 5, 6]
```

Las funciones sin receptor se convertirán a `(ParamTypeA, ParamTypeB, ...) -> ReturnType` donde `ParamTypeA` , `ParamTypeB` ... son el tipo de parámetros de la función y `ReturnType1 es el tipo de valor de retorno de la función.

```
fun foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
println(::foo::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo0, Foo1, Foo2) -> Bar
```

List(Recursividad)

Una lista es una colección ordenada de elementos. Para trabajar con listas inmutables se utiliza la interface List. Extiende de la interfaz Collection, y define una serie de funciones propias, entre las que podemos destacar:

→ `get(index: Int)`: elemento almacenado en el índice especificado.

→ `indexOf(element: E)`: índice de la primera ocurrencia del elemento pasado como un argumento en la lista, o -1 si ninguno es encontrado.
→ `lastIndexOf(element: E)`: índice de la última ocurrencia del elemento pasado como un argumento en la lista, o -1 si ninguno es encontrado.

→ `listIterator()`: iterador sobre los elementos en la lista.

→`subList(fromIndex: Int, toIndex: Int)`: lista que contiene la porción de la lista entre los índices especificados de inicio y fin.

En Kotlin tenemos diversos modos de crear listas:
→`listOf()` permite crear una lista inmutable. Podremos tener todos los elementos de un mismo tipo o de diferentes tipos de datos:

```
val primos: List<Int> = listOf(2, 3, 5, 7)
val nombres: List<String> = listOf("Juan", "Roberto", "María")
val listaMezclada = listOf("Juan", 1, 2.445, 's')
```

→ `emptyList()` crea una lista inmutable vacía.

```
val listaVacia: List<String> = emptyList<String>()
```

→`listOfNotNull()` crea una lista inmutable con solo elementos no nulos.

```
val listaNoNulos: List<Int> = listOfNotNull(2, 45, 2, null, 5, null)
```

→`arrayListOf()` crea una lista inmutable de tipo ArrayList:

```
val arrayList: ArrayList<String> = arrayLis-tOf<String>("un", "dos", "tres)
```

Para trabajar con listas mutables utiliza un desdendiente de MutableList. Esta interfaz extiende las interfaces MutableCollection y List.

La interfaz MutableList agrega métodos para la recuperación o sustitución de un elemento basado en su posición:

- `set(index: Int, element: E)`: sustituye un elemento en la lista, devolviendo el elemento que estaba previamente en la posición especificada.
- `add(index: Int, element: E)`: inserta un elemento.
- `removeAt(index: Int)`: elimina el elemento en un índice particular.

Para convertir una lista inmutable en mutable utiliza la función `toMutableList()`. Este método creará una nueva lista: Para crear una lista mutable de un cierto tipo desde cero, por ejemplo, de `String`, usamos `mutableListOf<String>()`, mientras que para tipos mixtos podemos solo usar la función `mutableListOf()` en su lugar:

```
val listaMutable: MutableList<String> = mutableLis-tOf<String>("uno","dos")
listaMutable.add("tres")
listaMutable.removeAt(1)
listaMutable[0] = "cuatro"
val mutableListMixed = mutableListOf("BMW", "Toyota", 1, 6.76, 'v')
```

Filter(Recursividad)

Devuelve una lista que contiene solo elementos que coinciden con el predicado dado.

```
inline fun <T> Array<out T>.filter(
    predicate: (T) -> Boolean
): List<T>

inline fun ByteArray.filter(
    predicate: (Byte) -> Boolean
): List<Byte>

inline fun ShortArray.filter(
    predicate: (Short) -> Boolean
): List<Short>

inline fun IntArray.filter(
    predicate: (Int) -> Boolean
): List<Int>

inline fun LongArray.filter(
    predicate: (Long) -> Boolean
): List<Long>
```

Devuelve un nuevo mapa que contiene todos los pares clave-valor que coinciden con el predicado dado .

```
inline fun <K, V> Map<out K, V>.filter(
    predicate: (Entry<K, V>) -> Boolean
): Map<K, V>
```

El mapa devuelto conserva el orden de iteración de entrada del mapa original.

```
val originalMap = mapOf ( "clave1" a 1 , "clave2" a 2 , "clave3" a 3 )

val filterMap = originalMap . Filtro { ella . valor < 2 }

println ( filteredMap ) // {key1 = 1}
// el mapa original no ha cambiado
println ( originalMap ) // {clave1 = 1, clave2 = 2, clave3 = 3}

val nonMatchingPredicate : (( Map . Entry < String , Int > )) -> Boolean = { it . valor == 0 }
val emptyMap = originalMap . filtro ( nonMatchingPredicate )
println ( emptyMap ) // {}
```

Bibliorafía.

- https://www.tutorialesprogramacionya.com/kotlinya/detalleconcepto.php?punto=36&codigo=36&inicio=30
- https://riptutorial.com/es/kotlin/example/4200/referencias-de-funciones
- http://www.androidcurso.com/index.php/99-kotlin/925-colecciones-en-kotlin-list-set-y-map?fbclid=IwAR2NRyAX6kUaPgQwnoGMgTOoYMUWjaVryaM4l0HpVBDE6O2tElzQnMfOHEY
- https://cursokotlin.com/capitulo-10-listas-en-kotlin/?fbclid=IwAR0SuZyvRpHX1hH-x-xwkH8KA-i2S2kxQ_Dqfc72yfc64QPzVq_Ay_YwvPI
- https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html