Funciones de orden superior.

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.

Una de las características del paradigma de la programación funcional son las 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.

Funciones como parámetros.

Se pueden pasar funciones como argumentos, a otras funciones. La función puede ser almacenada en una función o definirse en la llamada.

Veamos el siguiente ejemplo que multiplica dos números, pero no regresa el resultado, sino que llama a la función que le indiquemos con el mismo:

```kotlin
fun multiplicarYLlamar(n1: Float, n2: Float, callback: (Float) -> Unit ){
    val resultado = n1 * n2
    callback(resultado)
}

multiplicarYLlamar(10f, 4f, fun (resultado: Float){
    println("El resultado es: $resultado")
})
```

callback.kt hosted with ❤ by GitHub view raw

Aunque se vea complicado, esto es útil cuando usamos callbacks en Kotlin.

Fíjate en la sintaxis tanto de la definición, como de la llamada.

Al invocar a la función definimos la función de callback, pero podríamos haberla guardado en una variable y pasarla como argumento.

Se pueden pasar todo tipo de funciones como argumentos, con variables argumentos dentro de ellas, y así sucesivamente.

List (recursividad)

Aquí, exploraremos cómo la recursividad puede ser muy útil al implementar operaciones en listas (y colecciones, en general). En un artículo futuro, veremos cómo podemos implementar listas y operaciones en ellos de una manera puramente funcional y qué recursividad tiene para ofrecernos para ayudarnos a hacerlo de manera elegante (en Kotlin). Comenzaremos implementando una max()función, que devuelve el elemento máximo en una lista. Por supuesto, para que esto sea posible, los elementos de la lista deben ser comparables de alguna manera. Vea el código a continuación:

```
diversión <E> max (lista: Lista <E>): E? donde E: Comparable <E>
{// ---> (1)
   fun inner (l: List <E>, m: E): E {// ---> (2)
      return if (l.isEmpty ( )) m // ---> (3)
      más si (m <l [0]) interior (l. drop (1), l [0]) // ---> (4)
      más interior (l. drop (1), m) // ---> (5)
   }
   return if (list.isEmpty ()) null
   else inner (list. drop (1), list [0]) // ---> (6 )
}
```

Filter (Recursividad)

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: true significa que el elemento dado coincide con el predicado, false 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

```
 números  val =  listOf ( "uno" , "dos" , "tres" , "cuatro" )
val más  largo que3  =  números . Filtro { ella . longitud  >  3 }
println (más largo que3 )

val  numbersMap  =  mapOf ( "clave1"  a  1 , "clave2"  a  2 , "clave3"  a  3 , "clave11"  a  11 )
val  filteredMap  =  numbersMap . filtro {( clave , valor ) ->  clave . termina con ( "1" ) &&  val
println ( filteredMap )
```

La función básica de filtrado es filter(). Cuando se llama con un predicado, filter()devuelve los elementos de la colección que coinciden. Para ambos Listy Set, la colección resultante es una List, porque Maptambién es una Map.