Programació multimèdia
i dispositius mòbils

5. Funcions en Kotlin

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2021-22

Continguts

1 Funcions en Kotlin

Kotlin contempla que puguen existir funcions no lligades a objectes, com en altres llenguatges de programació com C o C++.

Aquestes funcions que definim dins un paquet kotlin, però fora d'una classe, objecte o interfície es coneixen com **funcions de primer ordre**, o ***Top-Level functions***, i poden ser invocades directament sense necessitat de crear un objecte.

En Java, fem ús de classes d'utilitat amb mètodes estàtics per proporcionar funcions no lligades necessàriament a objectes. En Kotlin, les funcions top-level ens donen esta funcionalitat.

1.1 Definició i invocació

Per tal de declarar una funció en Kotlin fem:

```
fun nomFuncio(paràmetre1:Tipus1, paràmetre2:Tipus2...):TipusRetorn{
    // Cos de la funció
}
```

Veiem algunes característiques sobre la declaració de funcions:

- Es declaren mitjançant la paraula clau `fun`,
- Els noms comencen amb minúscula, i s'expressen en *camelCase*,
- Els paràmetres de la funció s'especifiquen rere el nom, entre parèntesi, i de la forma `paràmetre:Tipus`. Aquests tipus s'han d'especificar necessàriament.
- El tipus de retorn de la funció pot indicar-se després del parèntesi amb la llista d'arguments, seguit de `:`.
- Quan la funció no torna cap valor significatiu, el seu *Tipus de retorn* per defecte és `Unit`, que seria l'equivalent a `void` en Java o C.

1.1.1 Valor per defecte

Kotlin també admet fer ús de valors per defecte en els paràmetres de les funcions, indicant aquest valor després de declarar el tipus:

```
fun nomFuncio(paràmetre1:Tipus1=Valor1, paràmetre2:Tipus2=Valor2)
```

Aquests valors per defecte s'utilitzaran quan no es proporcione l'argument a la funció.

```kotlin
fun saluda(nom: String = "món", salutacio:String = "Hola"): String {
    return "$salutacio, $nom!"
}
```

Interoperabilitat amb Java

Java no admet paràmetres per defecte als mètodes, pel que quan s'invoque a una funció Kotlin des de Java, s'haurien de passar tots es valors dels paràmetres. Kotlin ens ofereix l'anotació `@JvmOverloads` per tal de generar per nosaltres funcions/mètodes sobrecarregats que es puguen invocar des de Java.

L'exemple anterior de la funció `saluda`, en Java passaria a ser:

```java
String saluda();
String saluda(nom);
String saluda(nom, salutacio);
```

1.1.2 Invocació de funcions i pas de paràmetres

Per invocar una funció, només hem d'utilitzar el seu nom i passar-li entre parèntesi els arguments. El pas de paràmetres es pot fer de dos formes:

- Pas de paràmetres **posicional**, és a dir, la posició que ocupa l'argument en la definició (signatura) de la funció es correspon a la posició en la crida, com es fa en C o Java, i
- Pas de paràmetres **per nom**, és a dir, en la invocació a la funció s'indica el nom del paràmetre al què es fa referència.

Veiem alguns exemples per entendre millor el pas de paràmetres, com funcionen els valor per defecte i algunes restriccions al respecte:

- Pas de paràmetres **posicional** i **sense** utilitzar **valors per defecte**:

```kotlin
saluda("Pep", "Hello") // => Hello, Pep!
```

- Pas de paràmetres **posicional** i utilitzant **valor per defecte** per a la salutació:

```kotlin
saluda("Pep") // => Hola, Pep!
```

- Si volem utilitzar el **nom per defecte** i especificar la salutació, com que aquesta posicionalment es troba en segona posició, caldrà indicar els argumetns **per nom**:

```
saluda(salutacio = "Hello") // => Hello, món!
```

- També podem especificar els dos arguments per nom, encara que si estan en el mateix ordre que a la signatura de la funció, no caldria:

```
saluda(nom="Pep", salutacio = "Hello") // => Hello, Pep!
```

- Amb el posicionament per nom, podem variar l'ordre dels arguments:

```
saluda(salutacio = "Hello", nom="Pep") // => Hello, Pep!
```

- També podem mesclar **arguments de forma posicional**, i altres **amb nom**, sempre i quan els posicionals vagen davant:

```
saluda("Pep", salutacio = "Hello") // => Hello, Pep!
```

- Ara bé... **en el moment en què utilitzem algun paràmetre amb nom, tots els que indiquem al darrere, han de ser també amb nom**:

```
saluda(salutacio="Ieee!", "Pep")
error: mixing named and positioned arguments is not allowed
```

- Si quan definim una funció afegim un paràmetre per defecte abans dels paràmetres sense valors per defecte, aquest paràmetre per defecte només es podrà utilitzar si utilitzem la funció amb arguments amb nom. És a dir:

```

// si definim `saluda` amb el primer argument per defecte,
// però no el segon:

fun saluda(nom: String = "món", salutacio:String): String {
    return "$salutacio, $nom!"
}

// Per invocar la funció només amb l'argument "salutacio"
// i utilitzar el valor de nom de defecte, caldrà dir
// que és "salutacio" l'argument que enviem:

saluda(salutacio="Hello") // -->  Hello, món!
```

```
// Si passem només un argument sense nom,
// s'interpreta com el primer argument,
// i com que el segon no té valor per defecte
// donarà error:

saluda("Hello") // -> error: no value passed for parameter 'salutacio'
```

1.1.3 Llista d'arguments variable

Quan una funció ha de rebre una quantitat variable d'arguments, podem utilitzar la paraula clau `vararg` davant el nom de la variable, de manera que la funció accepte una llista de paràmetres separats per comes, que el compilador embolicarà en un array:

```
fun escriuLlista(vararg llista:String){
    for (item in llista) println (item);
}

escriuLlista("param1", "param2")
escriuLlista("param1", "param2", "param3", "param4")
```

En cas que una funció tinga combinats paràmetres d'altre tipus amb una llista, el que es fa habitualment és posar primer els arguments d'altre tipus, i deixar la llista per al final:

```
fun escriuLlista(desc: Boolean, vararg llista:String){
    when (desc){
    false -> for (item in llista) println (item);
    true -> for (i in llista.size-1 downTo 0) println(llista[i]);
    }
}
escriuLlista(false, "un", "dos", "tres") // --> un dos  tres
escriuLlista(true, "un", "dos", "tres") // --> tres  dos  un
```

En cas que volgam posar la llista en primer lloc, haurem d'utilitzar paràmetres amb nom per als altres valors:

```
fun escriuLlista(vararg llista:String, desc: Boolean){
    when (desc){
```

```
    false -> for (item in llista) println (item);
    true -> for (i in llista.size-1 downTo 0) println(llista[i]);
    }
}
escriuLlista("un", "dos", "tres", desc=false) // --> un dos  tres
escriuLlista("un", "dos", "tres", desc=true) // --> tres  dos  un
```

Quan no sabem a priori quins valors anem a passr en aquesta llista d'arguments -per exemple perquè els ha d'introduir l'usuari-, solem emmagatzemar aquests valors en estructures com un vector. Per tal de passar aquest vector a una funció que espera una llista variable d'arguments, el que farem és *estendre* aquest vector com a llista variable d'arguments, pel al què utilitzarem l'*operador d'estensió (spread operator)*: `*` davant el nom del vector:

```
val strArray=arrayOf<String>("un", "dos", "tres")

// Exemple incorrecte, no podem passar el vector directament
>>> escriuLlista(strArray, desc=true)
error: type mismatch: inferred type is Array<String> but String was expected

// Sinò que cal estendre aquest en forma de llista d'arguments variable:
>>> escriuLlista(*strArray, desc=true)
tres
dos
un
```

1.1.4 Retorn de valors múltiples

Quan necessitem que una funció retorne més d'un valor, podem englobar aquests en un objecte. Kotlin, ens ofereix, addicionalment els tipus de dades `Pair` i `Triple`, que retornen respectivament dos i tres valors, els quals no és necessari ni que siguen del mateix tipus:

```
// Aquesta funció realitza la divisió entre dos números i
// retorna un parell (Booleà, Enter). El primer valor indica
// si l'operació és o no correcta, i en cas que siga correcta
// el segon valor indicarà el resultat (null en cas contrari)

fun divisio(dividendo: Int, divisor:Int): Pair <Boolean?, Int?>{
    when(divisor){
```

```
        0 -> return Pair(false, null)
        else -> return Pair(true, dividendo/divisor)
    }
}

>>> divisio(3, 0)
res96: kotlin.Pair<kotlin.Boolean?, kotlin.Int?> = (false, null)
>>> divisio(3, 1)
res97: kotlin.Pair<kotlin.Boolean?, kotlin.Int?> = (true, 3)
```

En el cas del tipus *Pair*, podem utilitzar la funció *infix* `to` que compacta i fa més legible el codi:

```
fun divisio(dividendo: Int, divisor:Int): Pair <Boolean?, Int?>{
    when(divisor){
        0 -> return false to null
        else -> return true to dividendo/divisor
    }
}
```

En l'exemple anterior, potser no li trobem massa senit a la funció `to`, però si pensem en quan treballem amb dades de tipus *clau:valor* li podem trobar el sentit. Per exemple:

```
fun getHomeVar(): Pair <String?, String?>{
    when(System.getProperty("os.name")){
        "Linux" -> return "Linux" to "\$HOME"
        "Windows" -> return  "Windows" to "%USERPROFILE%"
        else -> return null to null
    }
}
```

1.1.5 Simplificant funcions

Quan una funció va a consistir només en una expressió, podem expressar-la de forma més compacta, sense le claus i rere el símbol =. Veiem per exemple com ho utilitzariem en una funció que converteix euros en l'equivalent en dólars:

```
fun euro2dollar(euro:Double):Double{
    return euro * 1.18;
}
```

Aquesta funció es podria haver simplificar en una línia amb:

```
fun euro2dollar(euro:Double):Double = euro*1.18
```

A continuació podem vore altre exemple per vore com saber si un número és parell:

```
// Aquesta funció retorna el resultat de l'expressió x%2==0
// i serà true si x és divisible per 2 i false en cas contrari
fun esParell(x: Int): Boolean = x % 2 == 0

// Quan el tipus del resultat es pot inferir de l'expressió,
// podem no incloure'l en la declaració:
fun esParell(x: Int) = x % 2 == 0
```

1.1.6 Funcions de primer ordre i Java

Com hem comentat, les funcions que definim fora de qualsevol classe, objecte o interfície, i que poden ser invocades directament, sense haver de crear cap objecte reben el nom de funcions de primer ordre o *top-level functions*.

Com que Java no suporta aquest tipus de funcions, el compilador de Kotlin genera una classe amb mètode estàtics, que es poden utilitzar des de Java. Aquesta és una pràctica habitual en les llibreríes d'utilitats.

Tenim el següent codi Kotlin:

```
// Indiquem el nom del fitxer
// que es generarà (Utils.class)
@file:JvmName("Utils")

fun test1():Unit{
    println("Funció de test")
}
```

Per tal d'utilitzar aquesta funció test1 des de Java, farem:

```
public class prova {
    public static void main(String[] args) {
        Utils.test1();
```

```
    }
}
```

Compilem al mateix directori i executem:

```
$ kotlinc Utils.kt
$ javac prova.java
$ java prova
Funció de test
```

Com podem veure, hem utilitzat funcionalitats definides en Kotlin des de Java, tal i com hem vist anteriorment que també és possible amb projectes Kotlin.

1.2 Expressions Lambda

Les *expressions lambda* (o funcion literals) tampoc estan lligades a classes, objecte o interfícies.

Aquestes funcions es poden passar com a arguments a altres funcions de tipus superior (*higher-order functions*, no confondre amb les *Top-Level functions* o funcions de primer ordre). Una expressió lambda representa el bloc d'una funció, i simplifica el codi.

En Java se suporten expressions lambda des de Java 8, però en Kotlin són lleugerament diferents.

Característiques de les expressions lambda:

- S'expresa entre {}
- No té la paraula clau `fun`
- No té modificadors d'accés (`private`, `public` o `protected`) ja que no pertany a cap classe
- És una funció anònima, que no té nom
- No espeficica el tipus de retorn, ja que és inferit pel compilador
- Els paràmetres no van entre parèntesi

A més, podem assignar una expressió lambda a una variable i executar-la.

1.2.1 Creació d'expressions lambda

En aquest apartat, anem a vore algunes formes de crear expressions lambda.

1. **Expressió lambda sense paràmetres i assignada a una variable.**
    - Declaració:

```
val msg = { println("Hola! Sóc una funció lambda") }
```

- Invocació:

```
msg()
```

Que és molt semblant a com ho fariem amb nodejs/javascript:

```
> var msg=function(){console.log("Funció en js");}
> msg()
```

2. **Exemple d'expressió lambda amb paràmetres:**

```
val msg = { cadena: String -> println(cadena) }
msg("Hola Kotlin!")
msg("Bon dia!")
```

Hem creat una expressió lambda que rep un paràmetre `cadena` indicant també el seu tipus (`cadena: String`). El símbol fletxa (`->`) serveix per sepaprar els paràmetres del cos de la funció, que va a la seua dreta. El tipus del paràmetre, també es podria ometre sempre que el compilador el puga inferir:

```
val msg = { cadena -> println(cadena) }
```

En cas de tindre diversos paràmeres, haurem de separar-los amb una coma, i no posar-los entre parèntesi.

```
val escriuSuma = { s1: Int, s2: Int ->
        println("Sumem $s1 i $s2")
        val result = s1 + s2
        println("La suma és: $result")
    }
```

1.2.2 Pas d'expressions lambda a funcions

Podem passar expressions lambda a altre funcions. Estes reben el nom de funcions d'ordre superior (higher-order functions), ja que són funcions de funcions, i poden rebre com a arguments tant expressions lambda com funcions anònimes.

Veiem un exemple. Les funcions `first()` i `last()` de les col·leccions de Kotlin retornen, respectivament, el primer i l'últim element d'una llista. Ambdues funcions accepten una expressió lambda com a paràmetre, i aquesta expressió rep un argument de tipus String. El cos d'aquesta funció serveix com a predicat per a buscar dins la col·lecció un subconjunt d'elements. Açò significa que l'expressió lambda és qui decideix quins elements de la col·lecció es tindran en compte quan es busque el primer i l'últim:

```
// Definim la llista
val llista: List<String> = listOf("un", "dos", "tres", "quatre", "cinc",
 ↪  "sis")
// Busquem el primer i l'últim element:
>>> llista.first()
res22: kotlin.String = un
>>> llista.last()
res21: kotlin.String = sis

// Afegim una funció lambda, que restringisca esta búsqueda
// a les cadenes de 4 caràcters:
>>> llista.first({ s: String -> s.length == 4})
res28: kotlin.String = tres

>>> llista.last({ s: String -> s.length == 4})
res27: kotlin.String = cinc
```

A més, si l'ultim argument de la funció és una expressió lamda, Kotlin ens permet eliminar els parèntesis:

```
llista.first{ s: String -> s.length == 4}
llista.last{ s: String -> s.length == 4}
```

I fins i tot, podem eliminar el tipus de paràmetre, ja que aquest serà sempre el mateix tipus que els elements de la col·lecció.

```
llista.first{ s -> s.length == 4}
llista.last{ s -> s.length == 4}
```

> **Exercici resolt**
>
> Donada la següent llista:

```
val llista2: List<String> = listOf(1, 2, 3, 4, 5, 6)
```

Com obtindríem l'últim número imparell i el primer número parell d'aquesta?

Solució

```
>>> llista2.first{ s -> s%2 == 0}
res42: kotlin.Int = 2
>>> llista2.last{ s -> s%2 == 1}
res43: kotlin.Int = 5
```

1.2.3 El nom d'argument `it`

Quan una funció lambda rep un únic argument i el seu tipus pot ser inferit, es genera automàticament un argument per defecte anomenat `it`, que podem utilitzar-lo per simplificar encara més les expressions. Per exemple:

```
llista.last{ it.length == 4}
llista2.last{ it%2 == 1 }
```

1.2.4 Return en expressions Lambda

Quan utilitzem un `return` en una expressió lambda dins una funció, aquest fa referència a la funció on es troba definida l'expressió, no a la pròpia funció lambda. Veiem-ho amb un exemple. Amb la següent funció volem mostrar els números d'una llista que no són divisibles per 2:

```
fun testReturn() {
    // Definim una llista d'enters
    val intList = listOf(1, 2, 3, 4, 5)
    // Recorrem aquesta llista amb la funció forEach.
    // Aquesta funció recorre tots els elements de la llista
    // i executa la funció lambda sobre cada element:
    intList.forEach {
        // Si l'element és divisible per 2, fem un return
        if (it % 2 == 0) return
        println("$it no és divisible");
    }
    println("Finalitzant testReturn")
}
```

```
>>> testReturn()
```

El resultat, després d'executar la funció només mostra:

```
1 no és divisible
```

Això es deu a que quan arriba al número 2, que sí que és divisible per 2, executa el `return`, i aquest, en lloc d'eixir de la funció *lambda*, ix de la funció `testReturn()`.

Si volem indicar que isca de l'expressió lambda, cal fer-ho explícitament fent ús d'etiquetes amb:

```
fun testReturn2() {
    val intList = listOf(1, 2, 3, 4, 5)
    intList.forEach {
        // Fem un return explícitament de forEach
        if (it % 2 == 0) return@forEach
        println("$it no és divisible");
    }
    println("Finalitzant testReturn2")
}
```

Ara sí:

```
>>> testReturn2()
1 no és divisible
3 no és divisible
5 no és divisible
Finalitzant testReturn2
```

A més, per altra banda, també podriem haver definit la nostra etiqueta:

```
 intList.forEach @tornaAci {
     if (it % 2 == 0) return@tornaAci
     ..}
```

Una altra solució a aquest problema sense utilitzar les etiquetes seran les funcions anònimes.

1.3 Funcions anònimes

Les funcions anònimes són altra forma de definir blocs de codi que passem a altres funcions. Les principals característiques d'estes funciosn són:

- No tenen nom,
- Es creen amb la paraula clau `fun`,
- contenen un cos de funció

Aquestes funcions, a diferència de les lambda, ens permetran indicar el tipus de retorn de la funció. Veiem-ho amb l'exemple de `last()`, on utilitzem una funció anònima en lloc d'una expressió lambda:

```
    llista.last(fun(cadena): Boolean {
     return cadena.length == 3
})
```

Amb funcions anònimes, quan utilitzem un return, sí que ix de la pròpia funció anònima, no de la funció que l'englobe, de manera que sí que podem expressar l'exemple del `forEach()` amb:

```
fun testReturn3() {
    val intList = listOf(1, 2, 3, 4, 5)
    intList.forEach (fun (num){
        // Aquest return farà referència
        // a la funció anònima
        if (num%2 == 0) return
        println("$num no és divisible")
        })
    println("Finalitzant testReturn3")
}
```

1.4 Funcions locals o nidificades

Per tal de modularitzar els nostres programes, a banda de la programació orientada a objectes, Kotlin ens ofereix les funcions locals (o nidificades), que són funcions declarades dins altres funcions, i que tindran sentit dins d'aquestes, no fora:

```
fun f1(){
    println("Dins de f1")
```

```
    fun f2(){
        println("Dins de f2")
    }
    f2()
}
>>> f1()
Dins de f1
Dins de f2
>>>

>>> f2()
error: unresolved reference: f2
```

Les funcions locals tenen accés a les variables i paràmetres de la funció dins la qual es defineixen, pel que no seria necessari passar estos arguments. Veiem-ho amb un exemple:

```
var a=1
fun f1():Int{
    var b=2
    fun f2():Int{
        var c=3
        // Des d'aci tenim accés
        // a les variables definides
        // fora de la funció.
        return (a+b+c)
    }
    return f2()
}

>>> println(f1())
6
```

1.5 Funcions d'extensió

Les funcions d'extensió ens permeten agregar noves funcionalitats a classes ja existents sense haver de crer una nova classe que herete de la que volem estendre. Aquestes funcions es declaren fora de les classes, pel que també són funcions de primer nivell (top-level). Kotlin, a més, també admet propietats d'extensió.

Per tal de crear una funció d'extensió ho farem com una funció comú, però afegint el nom de la classe al davant del nom de la funció. La clase sobre la que fem l'extensió s'anomenarà *tipus de receptor*,

i anomenarem *objecte receptor* a la instància de classe o el valor concret sobre el què s'invoca a la funció d'extensió.

Veiem un exemple:

```
// Estenem la classe String per comptar el
// número de vegades que apareix en ella
// un caràcter.
fun String.countChar(c: Char): Int{
    var count=0;
    for (i in this)
        if (i==c) count++
    return count
}

var a="Hola"     // Object Receptor
a.countChar('a') // -> res189: kotlin.Int = 1

a="babala"        // Object Receptor
a.countChar('a') // -> res192: kotlin.Int = 3
```

1.6 Funcions d'ordre superior o més alt (*High Order Functions*)

Com hem anticipat, una funció d'ordre superior és una aquella que:

- Rep altra funció (o expressió lambda) com a argument,
- Retorna una funció, o
- Ambdues coses.

A l'apartat sobre el pas d'expressions Lambda a funcions hem vist com passar expressions lambda a funcions d'ordre superior. Ara, el que vorem és ***com*** crear estes funcions.

Reprenent l'exemple de pas d'euros a dòlars de les expressions lambda, afegim altra funció lambda que convertisca d'euros a *yens*.

```
fun euro2dollar(euro:Double):Double = euro*1.18
fun euro2yen(euro:Double):Double = euro*124.68
```

A més, anem a definir una funció que anomenarem `euroChanger`, que rebrà com a primer paràmetre la quantitat en euros, com a segon paràmetre, l'expressió lambda que s'ha d'aplicar.

```
fun euroChanger(euro: Double, op: (Double) -> Double): Double {
    val result = op(euro)
    return result
}
```

Com veiem, aquesta funció retorna un tipus *Double*, el primer paràmetre que rep (*euro*) és també de tipus *Double*, i el segon paràmetre (*op*) es defineix com una funció que accepta un *Double* i retorna un *Double* (`op:(Double) -> Double`). Solem dir que aquest segon paràmetre és una funció *de Double a Double*. Una vegada dins la funció, podrem invocar a la funció que s'haja passat com a argument amb el nom d'aquest (*op*) i passar-li els argument que necessite.

Per tal d'invocar esta funció, cal passar-li com a segon argument una funció amb la mateixa signatura que definim per a l'argument (és a dir, de *Double a Double*). Per tal de passar la funció com a paràmetre, cal afegir al davant `::` i no utilitzar els parèntesis:

```
>>> euroChanger(12.0, ::euro2dollar)
res13: kotlin.Double = 14.16
>>> euroChanger(12.0, ::euro2yen)
res14: kotlin.Double = 1496.16
```

Per altra banda, també podem passar el cos de la funció en la pròpia invocació, i fer ús de l'argument autogenerat `it` de la següent manera:

```
>>> euroChanger(12.0, {it*0.9})
res18: kotlin.Double = 10.8
```

Retornant funcions

Com hem comentat, a més d'admetre funcions com a arguments, les funcions d'ordre més alt també poden retornar una funció. Per a això podem fer ús del nom d'una funció definida prèviament, i precedida dels `::`, o bé definir-la com a una funció lambda:

```
fun getFunc(divisa:String): (Double) -> Double {
    when(divisa){
    "dollar" -> return ::euro2dollar
    "yen" -> return ::euro2yen
    else -> return { it*0 }
    }
}
```

Ara ja podem utilitzar-la de la següent forma:

```
>>> euroChanger(12.0, getFunc("dollar"))
res33: kotlin.Double = 14.16
>>> euroChanger(12.0, getFunc("yen"))
res34: kotlin.Double = 1496.16
>>> euroChanger(12.0, getFunc("peseta"))
res35: kotlin.Double = 0.0
```

1.7 Tancaments (*Closures*)

Un tancament o *closure* és una funció que pot accedir i modificar variables definides fora del seu àmbit. En altres paraules, es té accés a variables que eren locals, però que ja no estan disponibles en l'àmbit.

La forma més senzilla d'exemplificar-ho és una funció que retorna una expressió lambda. Quan la funció està executant-se, l'expressió lambda té accés a les variables locals definides en la funció, però a més, té accés a elles quan la funció ha finalitzat, i per tant, les variables locals s'han destruït. Podem vore-ho com que la funció lambda *captura* les variables en el tancament.

Veiem-ho amb un exemple:

```
// Definim una funció que rep un enter
// i que retornarà una funció que no rep
// arguments i que no retorna res: ()->Unit
fun Comptador(ValorInicial:Int):()->Unit{
    // Definim una variable local a esta funció,
    // i la inicialitzem al valor que ens passen
    var a=ValorInicial
    // Aquesta lambda escriu el valor de la
    // variable "a" i la incrementa
    var lambda= {println(a);a=a+1}
    // El què es retorna és la funció lambda
    return lambda
}
```

Aquesta funció, bàsicament, ens retorna una funció lambda que no rep arguments ni retorna res, i que, quan la invoquem, escriurà el valor de la variable a i la incrementarà.

Per tal de veure'n el funcionament, invocarem a la funció `Comptador`, i ens guardarem el resultat en una variable:

```
var c=Comptador(0)
var d=Comptador(15)
```

Ara podrem invocar c com si fos una nova funció, que executarà el contingut de la lambda:

```
>>> c() // --> 0
>>> c() // --> 1
>>> c() // --> 2
>>> d() // --> 15
>>> d() // --> 16
```

Com veiem, tot i que la funció `Comptador` ja no *existeix*, el valor inicial que li hem donat per crear c i d queda eaccessible en aquestes variables que guarden la funció lambda.

Per altra banda, els tancaments poden treballar tant amb funcions anònimes com amb funcions lambda amb nom. Per tal d'utilitzar funcions amb nom, aquesta ha de ser una funció local i accedir a ella a través de l'operador `::`.

```
fun Comptador2(ValorInicial:Int):()->Unit{
    var a=ValorInicial
    // Traiem el contingut de la lambda
    // a una funció amb nom
    fun funcioAmbNom(){
        println(a);
        a=a+1
    }
    var lambda=::funcioAmbNom
    return lambda
}
```

L'ús de tancaments i funcions lambda cobra especial importància per definir funciona de *callback* als gestors d'esdeveniments. És a dir, funcions que s'executen en el moment en què es produeix determinat esdeveniment, com fer *click* o *tap* en un botó o rebre una resposta a una crida remota què hem fet. Aquestes funcions que s'executaran com a resposta a determinats esdeveniments (i que reben el nom de callbacks), han d'estar preparades per *respondre* a aquests esdeveniments, però no han de paralitzar la resta d'execució. Pensem per exemple en una aplicació que rep una llista d'items des d'Internet i els mostra per pantalla. Des que es fa la petició per demanar la llista fins que es rep, pot tardar una miqueta, i aquesta espera no deuria paralitzar l'aplicació. El que fem és definir una funció de callback que s'encarregue de *pintar* aquests items, i que s'executarà en el moment en què rebem la resposta.

Podeu trobar més informació sobre aspectes avançats de funcions en Kotlin a: https://code.tutsplus.com/es/tutorials/from-scratch-advanced-functions–cms-29534