Resum per a l'ús propi de Juanjo

pdf
html

Índex

1. Tema 0 . Programació Bàsica en Kotlin
    1. Consideracions generals
    2. Operadors interessants
    3. Instruccions condicionals
    4. Bucles
    5. Excepcions
    6. Definint classes
        1. Definint DTO o ValueObjects
    7. Mètodes ì propietats de classe o estàtiques en Java
    8. Herència
    9. Genèrics
    10. Enumeracions
    11. Funcions d'extensió
    12. Lambdas
        1. Definint HOF
        2. Clausures
    13. map-filter-fold
        1. Creant objectes anònims de consultes sobre objectes complexes
    14. Col·leccions

Tema 0 . Programació Bàsica en Kotlin

Consideracions generals

- Declaració de variables amb `var` (mutable) y `val` (inmutable)
- Tot s'ha d'inicialitzar menys si afegim la paraula reservada **Inicialització tardana**, afegint el modificador `lateinit`
- No existeix conversió implícita de tipus.

Operadors interessants

- `objecte?.data` : objecte es de tipus nullable i accedix a la data si el objecte no es null. En cas contrari tot s'avaluarà a null.
- `objecte!!.data` : objecte es de tipus nullable i el tracta com si fora no null. Si el fos llançaria una excepció.
- `o1 = Objecte = o2 ?: Objecte()` : `o1` es de tipus nullable i `o2` no. Si `o1` no es null tot se avalua tot a `o1` i si el fora a la part dreta que es una nova instància per defecte del tipus `Obejcte`

Instruccions condicionals

- Expression condicional equivalent al `string t = exp ? "true" : "false"` de C# `val t = if (exp) "true" else "false"`
- No existeix el `swicth` com instrucció i sols tenim `when` que se avalua com a expressió i si no retorna res com fa las voltes de una instrucció.

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

Bucles

```
for (i in 1..3) {
    println(i)
}

for (i in 6 downTo 0 step 2) {
    println(i)
}

while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

Excepcions

- `try` es una expressió:

```
val a: Int? = try { input.toInt() } catch (e: NumberFormatException) { null }
```

- Funcions que envolten excepcions (`Nothing`)

```
fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}

val s = person.name ?: fail("Name required")
println(s)     // 's' is known to be initialized at this point
```

Definint classes

Exemple de classe amb diferents definicions típiques...

```
// Definició de la classe i del constructor principal.
// Amés, hem definit implícitament, dues propietat titular.
class Compte(val titular: String, val número: Int) {

    // Propietat mutable privada sols para modificació inicialització.
    // Defineix un get i set auto-implementat com C#
    var saldo: Double = 0.0
        private set

    // Propietat pública mutable i annullable
    // en la que definim nosaltres el get i el set
    // filed: paraula reservada per fer referència al camp associat a la propietat.
    // value: paraula reservada per fer referència al valor rebut al setter.
    var banco: String? = null
        get() = field ?: "Desconegut"
        set(value) {
            field = value!!.toString()
        }

    // Propietat pública de solo lectura calculada i per tant (immutable)
    // També es por declarar així -> val hayDescubierto: Boolean = saldo < 0
    val hayDescubierto: Boolean
        get() = saldo < 0  // Definim el getter para la propietat

    // Constructor secundari recolzant en el principal.
    constructor(titular: String, número: Int, saldo: Double) : this(titular, número) {
        this.saldo = saldo
    }

    // Mètode públic normal
    fun ingress(quantitat: Double) {
        saldo += quantitat
    }

    // Invalidació ToString amb cos d'expressió
    override fun toString(): String = "Compte $número titular $titular saldo ${saldo}€"

    // Mètode públic que llança una excepció
    @Throws(IllegalArgumentException::class)
    fun reintegre(quantitat: Double) {
        if (quantitat > saldo) throw IllegalArgumentException("Saldo insuficient")
        saldo -= quantitat
    }
}
```

Definint DTO o ValueObjects

Si volem definir objectes amb tipus que facen de DTO o ValueObjcts com el tipus `record` en **C# 10** o **Java 17** tenim la possibilitat de definir-los com `data class`. Sus característiques són:

```
data class Empleado(val nombre: String, val edad: Int, val ciudad: Ciudad) {
    enum class Ciudad() { Elche, Alicante }
}

fun main() {
    val e1 = Empleado("Xusa", 45, Empleado.Ciudad.Alicante)
    val e2 = Empleado("Pepe", 54, Empleado.Ciudad.Alicante)
    val e3 = Empleado("Juanjo", 52, Empleado.Ciudad.Elche)
    val e4 = Empleado("Juanjo", 52, Empleado.Ciudad.Elche)
}
```

1. Defineixen por defecte `equals()`, `hashCode()` així com `==` i `!=`

   ```
   println(e3 == e4)   // mostra true en comptes de comparar referències
   ```

2. Defineixen por defecte `toString()`

   ```
   println(e3)         // mostra "Empleado(nombre=Juanjo, edad=52, ciudad=Elche)"
   ```

3. Ja que son Immutables tenim possibilitat de crear fàcilment copies amb `copy()`

   ```
   val e5 = e4.copy(ciudad = Empleado.Ciudad.Alicante)
   println(e4)         // Mostrarà "Empleado(nombre=Juanjo, edad=52, ciudad=Elche)"
   println(e5)         // Mostrarà "Empleado(nombre=Juanjo, edad=52, ciudad=Alicante)"
   println(e3 == e5)   // Mostrarà false
   ```

Mètodes ì propietats de classe o estàtiques en Java

Creem on `objecte global anònim` denominat per exemple `Datos`. En altres paraules, no estem definint una classe sinó un objecte instanciat, referenciat per el id `Datos` i de tipus anònim.

Després usarem la anotació `@JvmStatic` per a indicar que es un mètode o propietat de classe. Si no, posem `@JvmStatic` estarem definit un `Sigleton`.

```
object Datos {
    @JvmStatic
    val empleados = listOf(
            Empleado("Xusa", 45, Empleado.Ciudad.Alicante),
            Empleado("Pepe", 54, Empleado.Ciudad.Alicante),
            Empleado("Juanjo", 52, Empleado.Ciudad.Elche),
            Empleado("Vicente", 45, Empleado.Ciudad.Elche))
}

// Després Datos.empleados per accedir
```

Si volem fer-ho dins d'una altra classe marcaren el objecte creat amb el modificador `companion`

```
class MiClasse
{
    companion object Datos {
        val empleados = listOf(
                Empleado("Xusa", 45, Empleado.Ciudad.Alicante),
                Empleado("Pepe", 54, Empleado.Ciudad.Alicante),
                Empleado("Juanjo", 52, Empleado.Ciudad.Elche),
                Empleado("Vicente", 45, Empleado.Ciudad.Elche))
    }
}

// Després MiClasse.empleados per accedir (No fa falta possar Datos)
```

Herència

Paraula reservada `open` per permetre l'herència. En cas contrari la classe està segellada.

```
open class Persona(var nom: String, var edat: Int = 0) {
  ...
}

class Estudiant(
    nom: String,
    edat: Int = 0,
    var estudis: String) : Persona(nom, edat) {

}
```

Genèrics

Classes genèriques o parametritzades amb restriccions:

```
class ClasseGenerica<T: Comparable<T>>(t: T, c:String)
```

Mètode genèric amb cos de expressió. **La parametrització va davant del nom de la funció ja que el tipus va darrere.**

```
fun <T> funcioGenerica(param: T): T = param // Retorna la expressió
```

Enumeracions

Són classes com a Java

```
enum class CiclesInformatica(val valor: Int, val grau: String) {
    SMX(1, "Grau Mitjà"),       // Ordinal 0
    ASIX(2, "Grau Superior")    // Ordinal 1
}

val data : CiclesInformatica = CiclesInformatica.ASIX
println(data.ordinal); // Imprimeix 1
```

Funcions d'extensió

```
fun String.Escriu() = println(this);
fun main() { "Soc una cadena".Escriu() }
```

Lambdas

Són tipats com succeeix a C# i es defineixen sempre entre claus `{ definició }` anàlogament a altre llenguatges amb el operador `->` per example `{ v:Int -> v % 2 == 0 }`.

> 📌 **Nota:** Hem de posar el tipus del paràmetre si no va implícit.

Si es un procediment que no hi ha cap paràmetre de entrada no fa falta posar `{ () -> avaluació }` sinó directament `{ avaluació }`. Per això utilitzem las claus.

Poden definir-se de **tres formes**:

1. SAM ('*Single Abstract Method*') interface o també conegut com a '*functional interface*': Podem posar-li nom al tipus i tal i com succeeix a Java, es un interfície però amb una única funció. Per això posem `fun` davant de la paraula reservada `interface`.

```
fun interface Predicat<T> {
    fun compleix(dato: T): Boolean
}

fun main() {
    val esPar = Predicat<Int> { v -> v % 2 == 0 }
    println("Es par 4 = ${esPar.compleix(4)}")
}
```

2. El tipus es dedueix de la inicialització

```
fun main() {
    val esPar = { v: Int -> v % 2 == 0 }
    println("Es par 6 = ${esPar2(6)}")
}
```

3. El tipus es anònim i be definit per la signatura del lambda.

> 📌 **Nota:** Sols té sentit si anem a passar una funcions de ordre superior (HOF).

```
fun main() {
    // ací te poc sentit
    lateinit var esPar: (Int) -> Boolean
    esPar = { v -> v % 2 == 0 }
    println("Es par 9 = ${esPar(9)}")
}
```

Definint HOF

Podem utilitzar un tipus amb nom o no com ja hem indicat.

```
fun interface Predicate<T> {
    fun clumple(dato: T): Boolean
}

// En aquest cas el tipus del callback te nom i és Predicate<T>
fun<T> muestraPredicado1(nombre: String, dato: T,
                         predicadoCallback: Predicate<T>) {
    println("${nombre} ${dato} = ${predicadoCallback.clumple(dato)}")
}

// En aquest cas es tipus del callback és anònim i és (T) -> Boolean
fun<T> muestraPredicado2(nombre: String, dato: T,
                         predicadoCallback: (T) -> Boolean) {
    println("${nombre} ${dato} = ${predicadoCallback(dato)}")
}

// Important, tot i que podem passar el lambda com la resta de paràmetres.
// Kotlin ens recomana usar la següent sintaxi
// a on definim el lambda després de la cridada a la funció.
fun main() {
    muestraPredicado1("Es impar", 6) {
        v -> v % 2 != 0
    }
    muestraPredicado2("Es impar", 7) {
        v -> v % 2 != 0
    }
}
```

Clausures

Funcionen de la mateixa forma que a C#

```
// Es una HOF perquè retorna una funció
fun contador() : () -> Int {
    var i : Int = 0;

    // i es una variable clausurada.
    // s'obvia el () -> del tipus de retorn.
    return { i++ }
}

fun main() {
    var cuenta1 = contador()
    var cuenta2 = contador()

    println("cuenta1 = ${cuenta1()}")
    println("cuenta1 = ${cuenta1()}")
    println("cuenta1 = ${cuenta1()}")
    println("cuenta2 = ${cuenta2()}")
    println("cuenta2 = ${cuenta2()}")
}
```

map-filter-fold

```
val notas = listOf(1.0, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7);
val aprobados: Int = notas.map { n -> n.roundToInt() }
                          .filter { n -> n >= 5.0 }
                          .fold(0) { c, _ -> c + 1 }
println("Aprobados: $aprobados");
println("Aprobados: ${notas.map { n -> n.roundToInt() }.count { n -> n >= 5 }}");

val notaMayor: Int = notas.map { n -> n.roundToInt() }
                          .distinct()
                          .sortedByDescending { n -> n }
                          .first()
println("Nota mayor $notaMayor")
println("Nota mayor ${notas.map { n -> n.roundToInt() }.maxOf { n -> n }}")
```

Creant objectes anònims de consultes sobre objectes complexes

Suposem la següent definició de dades:

```
data class Empleado(val nombre: String, val edad: Int, val ciudad: Ciudad) {
    enum class Ciudad() { Elche, Alicante }
}
object Datos {
    @JvmStatic
    val empleados = listOf(
            Empleado("Xusa", 45, Empleado.Ciudad.Alicante),
            Empleado("Pepe", 54, Empleado.Ciudad.Alicante),
            Empleado("Juanjo", 52, Empleado.Ciudad.Elche),
            Empleado("Vicente", 45, Empleado.Ciudad.Elche))
}
```

Volem obtindre una llista de objectes anònims amb el nom i la ciutat dels empleats majors de 45 anys sense repeticions i ordenats per nom.

```
val res = Datos.empleados
    .filter { e -> e.edad > 45 }    // Filtrem per edat
    .map { e ->                     // Projectem lo filtrat a un objecte anònim
        object {
            val nombre = e.nombre
            val ciudad = e.ciudad
        }
    }
    .distinct()                     // Elimines repeticions
    .sortedBy { d -> d.nombre }     // Ordenem per nom
    .toList()                       // Passem la seqüència a una llesta

// Si volguérem retornar la llista
// deuríem usar una data class en comptes de objectes anònims.
println(res.joinToString("\n", String.format("%-10s%-10s\n","Nombre", "Ciudad")){
    d -> String.format("%-10s%-10s",d.nombre, d.ciudad)
})
```

Si volguérem mostrar el empleats per ciutat ordenats per nom podríem utilitzar el mètode `groupBy` com a C#

```
val empleatsPerCiutat : Map<Empleado.Ciudad, List<Empleado>> =
    Datos.empleados.groupBy { e -> e.ciudad }

var salida : StringBuilder = StringBuilder()
for (eXc in empleatsPerCiutat) {
    salida.append("${eXc.key}:\n")
    eXc.value.sortedBy { e -> e.edad }.forEach {
        e -> salida.append("\t${e}\n")
    }
}
println(salida);
```

Col·leccions

Tipus	Descripció	Literals	Mutabilitat	Modificar grandària
arrayOf	Array de objectes tradicional	Llista immutable	sí	no
listOf	Llista immutable	listOf(1, 2)	no	no
arrayListOf	Llista mutable	arrayListOf(1, 2)	sí	sí
mapOf	HashMap immutable	mapOf(1 to "A", 2 to "B")	no	no
mutableMapOf	HashMap mutable	...	sí	sí