Kotlin

MINI APUNTES DE KOTLIN 1

Programación Multimedia y Dispositivos Móviles

29 DE OCTUBRE DE 2024

ERIK AMO TOQUERO

Introducción a Kotlin

Kotlin es un lenguaje de programación muy similar a Java. De hecho, es Interoperable con este (es decir, puedes usar funciones de Java en Kotlin). Es un lenguaje multipropósito y multiplataforma. En general, tiene muchas cosas parecidas a Java, con ligeros cambios en la sintaxis.

Tipos de Datos en Kotlin

Para empezar, hay distintos Tipos de Variables en Kotlin:

- **Variables (var):** Cambia la información de su interior **sin cambiar su tipo.**
- **Constantes (val):** Una vez se inicializa, su valor no puede cambiar

También hay distintos Tipos de Datos:

- **Enteros (Int)**
- **Decimales (double)**
- **Decimales de coma flotante (float)**
- **Caracteres (Char)**
- **Cadenas de caracteres (String)**

Declaración de una variable: {var|val} nombre:[TipoDato[?]] [=...]

Los tipos de Datos se pueden comprobar con el operador '**is**'. La concatenación de Strings se puede hacer de 3 formas en este lenguaje:

- **Operador +:** *String1 + String2*
- **String Template ($):** *"$String1 $String2"*
- **String.format():** Igual que en java

Valores Nulos

En Java, puedes hacer simplemente Valor1 = null. En Kotlin no se puede, **pero** existe la posibilidad de hacer un **Tipo nulable**. ¿Qué es esto? Un valor que puede o no ser nulo, pero que ya tiene la opción de serlo. Se consigue poniendo '?' al final de la declaración de un atributo.

```
var valor:String? = null
```

Valor genérico

En Kotlin, cuando estamos creando una función o clase que tiene un valor cuyo tipo puede cambiar en función del contexto, se añade el valor genérico <T>. Este permite mutar el valor como el programa desee. Si quieres puede ser int en una ocasión y en otra puede ser un String

Estructuras de control

Como son muy parecidas en Java no me voy a parar a explicarlas mucho, salvo que sea explícitamente necesario

Entrada y Salida de Texto por línea de comandos

Más sencillo que java:

- print(): Muestra por pantalla sin saltos de línea
- println(): Muestra por pantalla con saltos de línea
- readln(): Guarda lo que el usuario escriba

Estructuras condicionales

Condicional simple – if

Como en java

```
if (condicion){
    [...]
}
```

Condicional doble – if-else

Como en java

```
if (condicion){
    [...]
} else {
    [...]
}
```

Grupos de condiciones – if-elseif-else...

```
if (condicion1){
    [...]
} else if (condicion2){
    [...]
} else {
    [...]
}
```

Grupos de condiciones – when(variable) -> switch

Es un switch en java, pero cambia un poco la sintaxis, puesto que aquí los casos deben ir entre corchetes si ocupan más de una línea y no hay breaks:

```
when (numero){
    1 -> {[...]}
    2 -> {[...]}
    [...]
    else -> {[...]}
}
```

Asignación de variables con when – valor = when(variable)

Sabiendo el uso en Java y la sintaxis when se saca:

```
var string:String = when (numero){
    1 -> {"Hola"}
    2 -> {"Adios"}
    [...]
    else -> {"Nada"}
}
```

Operador Ternario

Es un if-else resumido en lo que viene a ser una línea.
La sintaxis es valor = (if(condicion) caso_afirmativo else caso_negativo)

```
var saludo:String = (if(condicion) "Buenos dias" else "Buenas noches")
```

Bucles

Los bucles funcionan prácticamente igual que en Java, a excepción del for, el cual se parece más a Python

Bucle con condición inicial – while

Igual que en java

```
while (condicion) {
    [...]
}
```

Bucle con condición final – do-while

Igual que en java

```
do {
    [...]
}while(condicion)
```

Bucles for

Hay varios tipos de bucles for:

- Bucle for de **intérvalos**: Hace el bucle desde num1 hasta num2

```
for (i in num1..num2){
    [...]
}
```

- Bucle for de **iteradores**: Funciona en colecciones

```
for (i in lista){
    [...]
}
```

Funciones

Las funciones son bloques de código aparte del método principal, que se pueden llamar tantas veces como se decida y pasar distintos valores como parámetros.

Tiene dos tipos de sintaxis:

- Sintaxis general: fun nombre(Parámetros)[:Tipo devuelto]{instrucciones]

```
fun prueba(num1:Int, num2:Int):Int{
    return num1+num2
}
```

- Sintaxis en notación Pascal: nombre:tipo = valor por defecto

```
fun prueba(num1:Int, num2:Int):Int = num1+num2
```

También existen las **funciones con un número de parámetros indefinido**, lo cual será el último definido en el apartado de parámetros con la palabra clave 'vararg'

```
fun prueba(num1:Int, vararg num2:Int):Int{
    var numsumar:Int = 0
    for (num in num2){
        numsumar+=num
    }
    return num1+numsumar
}

fun main() {
    var suma = prueba(1,2, 3, 4, 5, 6, 7, 8)
    //El 1 es num1, el resto son todos los que forman el num2
}
```

Otros tipos de funciones son las **funciones genéricas**, las cuales utilizan un parámetro genérico. Con esto se consigue poder aplicar la misma función a distintos tipos de variables

```
fun <T>listarValores(vararg valores:T):List<T>{
    var lista:MutableList<T> = mutableListOf()
    for (valor in valores){
        lista.add(valor)
    }
    return lista
}

fun main() {
    println(listarValores(1, 2, 3, 4, 5, 6)) //T = Int
    println(listarValores("ASD", "DFG")) //T = String
    println(listarValores(true, false, true, false)) //T = Boolean
}
```

Usos de funciones:

- **Funciones locales:** Se utilizan dentro de otra función y usan los parámetros creados dentro de estas
- **Funciones miembros:** Funciones dentro de objetos de una clase

Funciones Lambda

Las funciones lambda, dicho de forma muy sencilla, son funciones que se pueden guardar en variables y usar como parámetro de otras funciones.

La declaración de estas es muy sencilla, ya que es una mezcla de la declaración de una función y una variable: {var|val} funcionLambda:(parametros)->Valor que devuelve = {}

```
var funcionLambda:(String)->Unit = { palabra ->
    println("$palabra")
}
```

El operador que, en este caso, se ha llamado 'palabra', hace referencia al String que recibe la función lambda como parámetro. Cuando sólo tiene un parámetro, se puede omitir esa palabra y usar el operador '**it**'

```
var funcionLambda:(String)->Unit = {println(it)}
```

Normalmente no necesitan return, ya que devuelven la última línea de la lambda, pero se le puede añadir un return poniendo return@nombrefuncion valor

Funciones de Orden Superior

Las funciones de orden superior son el motivo por el que las Lambda existen, puesto que son funciones que tienen otras funciones en su parámetro.

¿Por qué se utilizaría una función dentro de otra? Para reutilizar código. Puede que necesites exactamente la misma estructura de una función pero que se valide un dato de distinta forma. También es la forma que tienen todos los Listeners de todos los lenguajes de programación, puesto que se usa una función para asignar otra función a un escuchador.

```
fun funcionOrdenSuperior(lambda: (Int)->Boolean, num:Int){
    if (lambda(num)){
        [...]
    }
}
```

Esa es la forma que tienen las funciones de orden superior.

¿Dónde se supone que voy a usar esto? Un ejemplo es las funciones de filtro y orden de los Arrays, que utilizan este tipo de funciones:

```
fun filtrarLista(lambda: (Int)->Boolean, lista:List<Int>): MutableList<Int> {
    var listaNueva:MutableList<Int> = mutableListOf()
    for (i in lista){
        if (lambda(i)){
            listaNueva.add(i)
        }
    }
    return listaNueva
}
fun main() {
    var lista:MutableList<Int> = mutableListOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    var listaNueva:MutableList<Int> = filtrarLista({it%2==0}, lista)
    println(lista)
```

```
    println(listaNueva)
}
```

Función como tipo de dato

Es un tipo de función que permite usar un tipo de dato para usarlo de referencia en la función.

Para entenderlo, voy a hacer la explicación con el propio código:

```
fun String.verSiContieneOtroString(num1:Int, string: String):Boolean {
    return if (this.length > num1+string.length) {
        (this.substring(num1, num1+string.length) == string)
    } else {
        false
    }
}

fun verSiContieneieneOtroString2(stringExterior:String, num1:Int,
stringInterior: String):Boolean {
    return if (stringExterior.length > num1+stringInterior.length) {
        (stringExterior.substring(num1, num1+stringInterior.length) ==
stringInterior)
    } else {
        false
    }
}

fun main() {
    var st1 = "Hola"
    var st2 = "ol"
    var num = 1
    println(st1.verSiContieneOtroString(num, st2))
    println(verSiContieneieneOtroString2(st1, num, st2))
}
```

Las dos funciones hacen exactamente lo mismo, con la diferencia de que la primera (La función como tipo de dato) utiliza un tipo de dato para referenciarlo (es decir, solo se puede invocar desde un objeto de String en este caso). El segundo caso es el que hemos estado haciendo siempre. Ambos dan lo mismo y funcionan igual.

Esto es aplicable a Lambdas también

```
val lambda:String.(Int, String)->Boolean = {num1:Int,string:String->
    if (this.length > num1+string.length) {
        (this.substring(num1, num1+string.length) == string)
    } else {
        false
    }
}
```

Programación Orientada a Objetos

Las clases y el uso de objetos en Kotlin es muy similar a Java con ligeros cambios.

Definición de una Clase

Similar a Java: [propiedad] class nombreClase[(parámetros)]{ cuerpo }

```
class Persona(var nombre:String, var edad:Int){

}
```

La declaración de parámetros en la definición de la clase sirve para que, cuando se cree el **Constructor Principal**, es decir, lo mínimo indispensable para crear el objeto, pida esos valores.

Constructor Secundario

Si se da el caso de que hacemos más atributos y queremos hacer tanto objetos que lo usen como objetos que no, existe la posibilidad de crear un constructor a mayores que pida x valores:

```
class Persona(var nombre:String, var edad:Int){
    lateinit var apellidos:String
    constructor(nombre:String, apellidos:String) : this(nombre, 0){
        this.apellidos = apellidos
    }
}
```

Nota: Cuando haya un valor que no queramos iniciar al principio, se utilizará lateinit para iniciarlo cuando nos interese

Getter y Setter

En Kotlin, los getter y setter se forman ellos solos, pero el usuario puede hacer modificaciones si lo ve necesario:

```
class Persona(var nombre: String, var edad: Int){
    var apellidos:String = ""
        get() {
            return "Apellidos: $apellidos"
        }
        set(value:String){
            if (value.isEmpty()){
                field = "No puesto"
            } else {
                field = value
            }
        }
    constructor(nombre:String, apellidos:String) : this(nombre, 0,
""){
        this.apellidos = apellidos
    }
```

Creación de Instancias de una clase (Objetos)

Similar a Java pero con ligeros cambios. Uno de ellos es la desaparición de '**new**':

```
fun main() {
    var persona:Persona = Persona("Pedro",12) //Principal
    var persona2:Persona = Persona("Pedro","Perez") //Secundario
}
```

Herencia

La herencia en Kotlin se permite a partir del modificador '**open**'. Este modificador se puede utilizar tanto en métodos como en clases.

Definición de la clase padre: open class NombreClase[(parámetros)]{cuerpo de clase}

Definición de clase hijo: class NombreClaseHijo[(parámetros)]:NombreClase[(parámetros)]{cuerpo}

Este tipo de herencia solo permite hacer **sobrecarga** de valores y métodos.

```
open class Persona(){
    var nombre: String = ""
    open var edad: Int = 0 //Variable hereditable
    var apellidos:String = ""
        get() {
            return "Apellidos: $apellidos"
        }
        set(value:String){
            if (value.isEmpty()){
                field = "No puesto"
            } else {
                field = value
            }
        }
    constructor(nombre:String, apellidos:String) : this(){
        this.apellidos = apellidos
    }
    open fun hola(){ //Función hereditable
        println("Hola mundo")
    }
}

class Ninio():Persona(){
    override var edad: Int = 0 //Sobrecarga de edad
    override fun hola() { //Sobrecarga de función
        println("Adios mundo")
    }
}
```

La declaración de objetos de la clase padre se puede hacer de dos formas:

- Clásica: Llamar a la clase Padre para crear un objeto

```
var persona1:Persona = Persona()
```

- Polimorfismo: Instanciar un objeto de la clase padre llamando a la clase hija

```
var persona1:Persona = Ninio()
```

Encapsulamiento

Esto trata simplemente de los modificadores que afectan a las clases, métodos y propiedades de Kotlin. Hay 4 modificadores de visibilidad:

- **public:** Se puede acceder a la clase o método desde cualquier lado
- **private:** Solo se puede acceder desde la misma clase
- **protected:** Solo se puede acceder desde la clase principal o sus subclases
- **internal:** Puedes acceder a ellos si están en el mismo módulo (Dentro del mismo archivo o carpeta)

En el caso de poner un modificador a las variables, sus getter y setter tienen que tener el mismo modificador.

Los métodos constructores también pueden tener modificadores.

Modificador	Accesible en la misma clase	Accesible en subclase	Accesible en el mismo módulo	Accesible fuera del módulo
private	✓	X	X	X
protected	✓	✓	X	X
internal	✓	✓	✓	X
public	✓	✓	✓	✓

Interfaces

Funcionan igual que en Java. Sirven a modo de contrato de qué mínimo de funciones debe tener implementada una clase.

```
interface Matematicas {
    fun suma(num1:Int, num2:Int):Int
    fun resta(num1:Int, num2:Int):Int
    fun multiplica(num1:Int, num2:Int):Int
    fun divide(num1:Int, num2:Int):Int
}

class Calculadora:Matematicas{
    override fun suma(num1: Int, num2: Int): Int {
        return num1 + num2
    }
    override fun resta(num1: Int, num2: Int): Int {
        return num1 - num2
    }
    override fun multiplica(num1: Int, num2: Int): Int {
        return num1 * num2
    }
    override fun divide(num1: Int, num2: Int): Int {
        return num1 / num2
    }
    fun resto(num1: Int, num2: Int): Int {
        return num1%num2
    }
}
```

Abstracción – Herencia 2.0

Las clases abstractas son aquellas clases que no se pueden instanciar directamente en un código, sino que se instancian por medio de sus clases hijas.

La abstracción crea una relación entre clases padre e hija, haciendo las subclases (clases hija) hereden todos los métodos y atributos de la superclase (clase padre). Cuando la clase padre tiene un atributo o un método abstracto, se sobrecargará por medio de **override** en la clase hija.

Definición de clases padre: abstract class nombrePadre[(parámetros)]{}

Definición de clases hija: class nombreHija[(parámetros)]:nombrePadre[(parámetros)]{}

```
abstract class ClasePadre(var edad:Int){
    var nombre:String = "Nada"
    abstract fun saludar();
    fun despedirse(){
        println("Adios")
    }
}
class ClaseHija(edad: Int):ClasePadre(edad){ //Se pasa como parámetro
los que tenga el padre
    override fun saludar() {
        println("Hola")
    }
}
fun main() {
    val hija:ClaseHija = ClaseHija(12);
    println(hija.edad) //Muestra la edad
    hija.saludar() //Función sobrecargada
    hija.despedirse() //Puede acceder a funciones del padre
    println(hija.nombre) //Puede acceder a variables declaradas en el
padre
}
```

Enumeración – Clase enum

Las clases enum o clases type-safe son clases que toman valores predefinidos, los cuales luego puede usar el sistema.

Declaración: enum class nombreClase[(parámetros)]{valores**;** funciones}

Es importante que, si el enum hace algo más que marcar x valores, se separe las funciones de los valores con un ‘;’

```
enum class Colores{
    ROJO, AZUL, VERDE, AMARILLO
}
```

Clases Anidadas y Clases Internas

Como concepto, son exactamente lo mismo: Una clase creada dentro de otra. Eso sí, tienen funciones distintas:

Clase Anidada:

Una clase dentro de otra la cual solo puede acceder a sus métodos y variables:

```
class ClasePrincipal(var nombre:String, var edad:Int){
    class ClaseAnidada{
        fun saludar(){
            println("Hola Mundo")
        }
    }
}
fun main() {
    val claseAnidada:ClasePrincipal.ClaseAnidada =
ClasePrincipal.ClaseAnidada()
    claseAnidada.saludar() //Esto es lo único que puede hacer
}
```

Clase Interna

Una clase dentro de otra, pero la interna sí que puede acceder al contenido de la externa:

```
class ClasePrincipal(var nombre:String, var edad:Int){
    inner class ClaseInterna{
        fun saludar(){
            println("Hola! Soy ${this@ClasePrincipal.nombre} y tengo
${this@ClasePrincipal.edad} años")
        }
    }
}
fun main() {
    val claseInterna:ClasePrincipal.ClaseInterna =
ClasePrincipal("Pedro", 12).ClaseInterna()
    claseInterna.saludar()

    //Se puede hacer esto también:
    val clasePrincipal = ClasePrincipal("Jose", 19) //Primero la
principal
    val claseInterna2 = clasePrincipal.ClaseInterna() //Después la
interna
    claseInterna2.saludar() //Saludar
}
```

ese this@ClasePrincipal.nombre se pone porque, desde la clase interna, está llamando a la instancia que la ha creado, su clase superior por así decirlo. Por eso se puede hacer tanto con una clase nueva como con una ya creada, porque son 2 clases distintas como tal.

Clases Anónimas – Object Expressions

Son instancias de clases sin nombre, las cuales no admiten definición de constructores. Son útiles cuando se necesita implementar una interfaz o una clase abstracta pero no necesitamos crear una clase, más que nada por ahorrar código.

Su definición es object:tipoClase(){funciones}

Un ejemplo que hemos usado es con la declaración del listener de un TextField, ya que necesitamos 3 funciones distintas en un solo escuchador:

```
editText.addTextChangedListener(object:TextWatcher{
    override fun beforeTextChanged(p0: CharSequence?, p1: Int, p2:
Int, p3: Int) {
        TODO("Not yet implemented")
    }

    override fun onTextChanged(p0: CharSequence?, p1: Int, p2: Int,
p3: Int) {
        TODO("Not yet implemented")
    }

    override fun afterTextChanged(p0: Editable?) {
        TODO("Not yet implemented")
    }
})
```

Data Class

Son clases creadas para almacenar distintos datos. Trae unos métodos por defecto: toString(), equals() y copy()

Los Data Class permiten dividir un objeto en valores de sus atributos (**desestructuración**), llamando a funciones de tipo ComponenteN para acceder a los datos de las propiedades, donde N será desde el primer valor hasta el último de la Data Class.

```
data class PruebaDatos(var nombre:String, var apell:String, var
edad:Int){
}
fun main() {
    var prueba1:PruebaDatos = PruebaDatos("Jose", "Paco", 12)
    println(prueba1.toString()) //ToString
    var prueba2 = prueba1.copy() //Copy
    prueba2.apell = "Pérez"
    println(prueba1.equals(prueba2)) //Equals
    val (atrib1, atrib2) = prueba1 //Guarda el nombre y el apellido
    println("$atrib1 $atrib2")
}
```

Type Alias:

Es una forma de nombrar a los tipos ya existentes.

Imaginemos por un momento que tenemos una función lambda que se repite muchas veces y es larguísima, como es (Int, String, Boolean, Char) -> Boolean. Para ahorrarnos poner todo eso cada vez que vayamos a usar ese tipo de función, usamos el typealias:

Su declaración es typealias nombreTipo = tipo

Esto sirve para funciones anidadas, funciones y variables

```
class ClasePrincipal(var nombre:String, var edad:Int){
    inner class ClaseInterna{
        fun saludar(){
            println("Hola! Soy ${this@ClasePrincipal.nombre} y tengo
${this@ClasePrincipal.edad} años")
        }
    }
}

typealias Interna = ClasePrincipal.ClaseInterna //Con una clase interna
typealias TipoLargo = (Int, Boolean, String, Char) -> Boolean //Con una
función
typealias Frase = String //Con un tipo de variable

fun main() {
    var funcionInterna:Interna = ClasePrincipal("AA", 12).ClaseInterna()
    var lambdaLarga:TipoLargo = { num1:Int, bool:Boolean, st:String, char:Char
->
        bool && st[num1] == char
    }
    var frase:Frase = "Hola"
}
```

Desestructuración

Para explicar un poco mejor lo hecho en las data class, la desestructuración se basa en guardar las variables de un objeto en distintas variables. Se puede hacer de un objeto ya creado o de una función que retorne un objeto. Va guardando los datos en el orden del constructor del objeto, pudiendo saltar alguno con '_'.
Se puede hacer en mapas para separar clave y valor

```
data class PruebaDatos(var nombre:String, var apell:String, var
edad:Int){
}
fun main() {
    val prueba1:PruebaDatos = PruebaDatos("Jose", "Paco", 12)
    val (atributo1, atributo2) = prueba1 //Guardo los valores del nombre y el
apellido
    val (atributo3, _ , atributo4) = prueba1 //Guardo el nombre y la edad
    val mapa:Map<Int, String> = mapOf(Pair(1, "A"), Pair(2, "B"), Pair(3,
"C"))
    for ((clave, valor) in mapa){ //Ejemplo en mapa
        println("$clave $valor")
        println("${mapa[clave]}")
    }
}
```

Extensión

Son funciones que amplian los miembros de una clase ya creada. Sigue la estructura de las **funciones como tipo de dato** (pg 7):
fun clase.nombreFuncion([parámetros]):tipoRetorno{}

```
class Persona(var nombre:String, var apellidos:String, var edad:Int){
    fun saludar(){
        println("Hola")
    }
    fun despedirse(){
        println("Adios")
    }
}


fun Persona.calculosRandom():Int{
    return this.nombre.length - this.apellidos.length
}
```

Se puede usar sin problema la nueva función con los objetos de tipo Persona en este caso.

Colecciones

Las colecciones funcionan de distinta forma en Kotlin, ya que, además de los tipos de colecciones, hay otra distinción más:

- **Colección Mutable:** Permite cambiar su tamaño a lo largo del programa con las funciones de adición y eliminación (add/put y remove)
- **Colección Inmutable:** No permite cambiar su tamaño

Quitando esa distinción, hay 3 tipos de funciones principales:

- **List:** Colección ordenada de elementos por índices.
- **Set:** Colección de elementos que no permiten repetidos
- **Map:** Pares clave-valor: Las claves no permiten repetición mientras que los valores sí.

Todas las colecciones permiten el uso de la función de orden superior **forEach** y el uso del operador **in** en funciones.

Listas

Conjuntos ordenados con índices de 0 hasta el tamaño-1 de la lista

Tiene los métodos comunes **indexOf(valor)**, que muestra la posición del primer valor de la lista con ese dato (-1 si no encuentra nada), y **size**, que muestra el tamaño de la lista

Las **interfaces mutables** se instancian usando **mutableListOf(lista)** y puede usar funciones como add, remove, removeAt...

Las **interfaces inmutables** se instancian usando **listOf(lista)** y no permite modificar el tamaño, por lo que no incluye add, remove, etc.

```
fun main() {
    var listaMutable:MutableList<String> = mutableListOf("Hola",
"Adios")
    var listaInmutable:List<String> = listOf("Hola", "Adios")
    println(listaMutable.size) //Muestra el tamaño
    println(listaInmutable.indexOf("Hola")) //Muestra el indice
    listaMutable.add("Buenas") //Añade buenas al mutable
    listaMutable.removeAt(2) //Quita el de la tercera posicion (3-1)
}
```

Sets

Conjuntos sin orden que no admiten valores duplicados, considerando duplicados también a colecciones que tengan el mismo tamaño y elementos.

Tienen las mismas funciones que las listas (salvo las funciones que usen numeración) y usan más memoria que estas, pero son útiles para garantizar singularidad.

```
fun main() {
    var listaMutable:MutableSet<String> = mutableSetOf("Hola",
"Adios")
    var listaInmutable:Set<String> = setOf("Hola", "Adios")
    println(listaMutable.size) //Muestra el tamaño
    println(listaInmutable.indexOf("Hola")) //Muestra el indice
    listaMutable.add("Buenas") //Añade buenas al mutable
    listaMutable.remove("Buenas")
}
```

Mapas

Conjuntos clave-valor. Si es mutable, permite añadir datos nuevos con put(clave, valor) o quitarlos con remove(clave)

Para acceder a información de un mapa, se hace con nombreMapa[clave]

```
fun main() {
    var mapa:MutableMap<Int, String> = mutableMapOf()
    mapa.put(123, "AAA")
    mapa[123] = "BBB"
    mapa.remove(123)
}
```