Tema 0 . Programación Básica en Kotlin

Índice

Introducción

Kotlin es un lenguaje de programación fuertemente tipado desarrollado por **JetBrains** en 2010 y que está influenciado por lenguajes como **Scala , Groovy y C#**.
La mayoría de desarrollos con Kotlin, tienen como destino Android o la máquina virtual de java (JVM), y aunque también ofrece la posibilidad de desarrollos para la máquina virtual de JavaScript o código nativo, por ahora todavía son pocos los programadores que lo usan para estos destinos.
A partir de la actualización Kotlin 1.3.30, se incluyeron las mejoras para Kotlin/Native que permite compilar el código fuente de Kotlin en datos binarios independientes para diferentes sistemas operativos y arquitecturas de CPU, incluido IOS, Linux, Windows y Mac.

Variables

Igual que en todos los lenguajes de programación, en Kotlin también tendremos el recurso de las variables para almacenar valores. En Kotlin nos podemos encontrar con variables **inmutables** y variables **mutables**. En el caso de las primeras, una vez se le asigna un valor a la variable no podrá ser modificado, es decir, se comporta como una constante (lo mismo que utilizar final en Java o const en C#). Mientras que con la mutables podremos modificar en cualquier momento el valor de la variable.
Para declarar una variable como mutable, la tendremos que preceder de la palabra clave **var** mientras que para las inmutables usaremos **val**.
El concepto de inmutabilidad es muy interesante. Al no poder cambiar los objetos estos son más robustos y eficientes. Siempre que podamos hemos de usar objetos inmutables.

```
//Variables mutables
var mutable:Int=5;
mutable+=7;
var numeroDecimales=3.14F;
numeroDecimales=3.12F;
//Variables inmutables
val inmutable:Char='C';
inmutable='A'  //Error compilación!! no se puede modificar una variable inmutable
```

> Como se puede ver en el ejemplo, cuando declaramos una variable, podemos indicar el tipo de esta o esperar a que el compilador lo infiera.

Para definir el tipo, tendremos que indicarlo con `:tipo` después del nombre:

(var|val) nombreVariable [:tipo][=valor]

Los tipos básicos de variables en Kotlin son:

Tipo	Valor	Tamaño
Byte	5.toByte()	8 bits
Short	5.toShort()	16 bits
Int	5	32 bits
Long	5L	64 bits
Float	1.45F	32 bits
Double	1.45	64 bits
Boolean	true	
Char	'H'	16 bits
Unit	Unit	0 bits

> El tipo `Unit` corresponde a void en Java y C#

Cuando una **variable mutable** declarada en el cuerpo de una clase, no se quiere inicializar en el momento de la declaración, existe el concepto de `Inicialización tardía`, añadiendo el modificador **lateinit** delante de la declaración de la variable.

```
public class Ejemplo{
lateinit var cadena: String
fun miFuncion() {
  cadena = "Hola Mundo";
  println(cadena);}
}
```

> ✋ Como en otros lenguajes de programación, se debe controlar las operaciones con variables de distintos tipos, para evitar resultados inesperados o incluso excepciones. Kotlin tiene varios métodos **toX()** para cambiar los valores al tipo que necesites:

```
fun main (args: Array <String>) {
   var a:Int
   var b=3.5f
   a=b+2
   println(a);
}
```

> Produciría el error ***error: type mismatch: inferred type is Float but Int was expected.***
> Mientras que: `a=b.toInt()+2` , evitaría el error, aunque la función redondearía a 3.

String

El tipo String representa el literal de cadena ya conocido, podemos ver un ejemplo del uso de templates `$` para la salida por pantalla.

```
fun main (args: Array <String>) {
val cadena = "El resultado de:"
val a = 2
val b = 5
println("$cadena $a + $b es: ${a + b}")
}
```

```
El resultado de: 2 + 5 es: 7
```

Control de flujo

Las instrucciones **if/else, for y while** podemos considerar que tienen un uso similar al de otros lenguajes, así que vamos a explicar solamente los elementos que lo diferencian:

for y while

Se diferencia sobre todo en que se tienen que usar rangos en la sentencia for y se pueden usar en la while:

```
fun repeticionConFor(v:Int)
{
   for(i in (0..v).reversed()) println("$i")
}
fun main() { repeticionConFor(10)}
```

```
fun repeticionConFor(tope:Int)
{
   var contador:Int=0
   do{
         var numero=readLine()!!.toInt();
         contador++;
   }while(contador<tope && numero!in 1..100)
}
fun main() { repeticionConFor(10)}
```

when

Sustituto de **swich**, y que también lo podemos encontrar similar en C#. Con un ejemplo quedaría explicado:

```
fun getEstacion(entrada:Int):String
{
return when(entrada)
{
    1-> "primavera"
    2-> "verano"
    3-> "otoño"
    4-> "invierno"
    else -> "Estación incorrecta"
}
}
fun main() { println(getEstacion(2))}
```

Otro ejemplo de la gran funcionalidad que nos permite la sentencia **when** podría ser el siguiente. Observa que en este caso el when es **sin argumentos**:

```
fun noHagoNada(x:Int, s:String, v:Float):String
{
   val res = when {
      x in 1..10 -> "entero positivo menor de 10"
       s.contains("cadena") -> "incluyo cadena"
       v is Comparable<*>-> "Soy Comparable"
       else -> ""
   }
   return res
}
fun main() { println(noHagoNada(2, "Hola", 3.5f));}
```

Funciones

Si queremos crear una función en kotlin tendremos que precederla de la palabra reservada **fun**. Por tanto, una función constará de la palabra fun seguida por el nombre de la función y entre paréntesis los parámetros, siempre y cuando los tenga. Si la función retorna un valor se definirá al final de la signatura (esto último se puede omitir siempre que la función no devuelva nada).

fun nombreFuncion([nombreVariable:Tipo, nombreVariable:Tipo, ...]):TipoDevolucion

```
fun sumaDatos(datoUno:Int, datoDos:Char )
{
   println("${datoUno + datoDos.toInt()}");
}
fun main() {
   var mutable:Int=5;
   mutable+=7;
   var numeroDecimales=3.14F;
   numeroDecimales=3.12F;
   val inmutable:Char='C';
   sumaDatos(mutable,inmutable);
}
```

Con valor de retorno:

```
fun mayor(numeroUno:Int, numeroDos:Int):Int
{
 return if(numeroUno>numeroDos)  numeroUno else numeroDos
}
fun main() { println(mayor(5, 7));}
```

Si la función solo tiene una instrucción, como en el caso anterior, Kotlin permite omitir las llaves.

```
fun mayor(numeroUno:Int, numeroDos:Int):Int= if(numeroUno>numeroDos)  numeroUno else num
```

Funciones de Extensión

Kotlin al igual que otros lenguajes, nos permite extender la funcionalidad de clases existentes, agregando funciones a estas. En el siguiente ejemplo, se ha añadido la función Escribe() a la clase String, por lo que se podrá usar con objetos de este tipo.

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

Arrays y Colecciones

En Kotlin hay diferentes formas de generar **listas de objetos**: arrays, listas inmutables, listas mutables, etc. Por un lado hay que distinguir la inmutabilidad de la colección de la inmutabilidad de la variable que la contiene.

	arrayOf	listOf	ARRAYLISTOF	LINKEDLISTOF

	arrayOf	listOf	ARRAYLISTOF	LINKEDLISTOF
Descripción	Array de objetos tradicional	Lista inmutable	Lista mutable	Lista enlazada mutable
Literales	arrayOf(1, 2, 3)	listOf(1,2, 3)	arrayListOf(1, 2, 3)	linkedListOf(1, 2, 3)
Inmutabilidad	no	sí	no	no
Modificar longitud	no	no	sí	sí

> ✋ En Kotlin las declaraciones de arrays no llevan corchetes como en otros lenguajes, sino que se utilizan los métodos de la clase Array.
> En los siguientes ejemplos podemos ver diferentes formas de recorrer el array mediante un bucle for.

```
fun main()
{
    val weekDays = arrayOf("primavera", "verano", "otoño", "invierno")
    for (dato in weekDays)  println(dato)
}
```

Además Kotlin distingue los arrays de primitivas usando clases propias: `BooleanArray, ByteArray, CharArray, ShortArray, IntArray, FloatArray, DoubleArray`, y la forma de construir literales es mediante: booleanArrayOf, byteArrayOf, charArrayOf, shortArrayOf, intArrayOf, floatArrayOf, doubleArrayOf.

Clases

Todos los elementos en Kotlin son públicos por defecto, por lo que también lo serán las clases. Las clases nos sirven para referenciar objetos del mundo real, y mediante las propiedades podemos definir las distintas características que nos interese manipular con las cuales conste ese objeto; por ejemplo, para definir una clase Persona en Kotlin con algunas propiedades (sí, propiedades y no atributos) podemos hacer lo siguiente: