# MANUAL BÁSICO DE KOTLIN

| Información general | |
|---|---|
| **Duración estimada en minutos:** | 120 |
| **Docente:** | Carlos Andrés Florez Villarraga |
| **Guía no.** | 02 |

## Información de la Guía

### KOTLIN

Kotlin es un lenguaje de programación creado en 2010 por Jetbrains. Este lenguaje permite hacer que el desarrollo Android sea más fácil, rápido y mucho más agradable.

Kotlin es un lenguaje moderno, estáticamente estandarizado, compatible con Android que soluciona muchos problemas de Java, como las excepciones de puntero nulo o la excesiva verbosidad de código. Kotlin es un lenguaje inspirado en Swift, Scala, Groovy, C # y muchos otros lenguajes. Kotlin se esfuerza por no repetir los errores de otros idiomas y aprovechar sus funciones más útiles. Cuando se trabaja con Kotlin, realmente se puede sentir que este es un lenguaje maduro y bien diseñado. Las ventajas principales de este moderno lenguaje de programación son:

- Seguro contra nulos.
- Ahorro de código.
- Características de programación funcional (higher-order functions, function types y lambdas).
- Fácil de usar.

A continuación se listan algunos ejemplos de las funcionalidades más importantes al momento de programar usando Kotlin.

Cada uno de los ejemplo que se prueban a continuación se puede probar en el compilador online: https://play.kotlinlang.org/

### Variables

Kotlin es un lenguaje no tipado, por lo cual se pueden declarar variables de la siguiente forma:

```
var algo = 1
var algo = true
var algo = "Prueba"
var algo = 1.2
var algo = 'A'
```

Sin embargo, si se desea tipar la variable también se puede usando la siguiente sintaxis:

```
var algo1:Int = 1
var algo2:Boolean = true
```

```
var algo3:String = "Prueba"
var algo4:Float = 1.2f
var algo5:Char = 'A'
```

Los tipos de datos de Kotlin son equivalentes a los existentes en Java, pero con pequeños cambios en los llamados. También existe el tipo de dato `Any` para definir que puede ser cualquier tipo.

Para la declaración de una variable sin la necesidad de inicializarla se hace lo siguiente:

```
var resultado:Int
```

También se puede usar la palabra reservada "*val*" en lugar de "*var*" para hacer que el valor almacenado no sea mutable, lo que es equivalente a las constantes en Java.

**Operadores y expresiones**

Los operadores y las expresiones aritmética relacionales y lógicas son equivalentes a las de Java por los cual se da por hecho el conocimiento de estas. Aunque cabe resaltar que la comparación de posiciones de memoria se debe realizar usando el operador "===" (con tres signos igual).

Para más información sobre las igualdades:

https://kotlinlang.org/docs/reference/equality.html

**Concatenación**

La concatenación en Kotlin solo necesita de agregar dentro de una cadena las variables deseadas acompañadas del signo "$", tal como se muestra a continuación:

```
var cadena1:String = "Hola "
var cadena2 = "Mundo"
var anio = 2017
var saludo ="$cadena1 $cadena2 $anio!!!"
```

**Conversión de tipo (casting)**

Para la conversión de tipo, todos los tipos de datos básicos manejan la función *toTipoDato*, por ejemplo:

```
var numero:Int = 1
var nuevo = numero.toString()
```

Con el uso de la función `toTipoDato()` se pueden hacer el *casting* entre los diferentes tipos de datos básicos.

**Condiciones con if**

Los condicionales *if* en Kotlin son iguales que en Java. Un ejemplo es la siguiente condición simple:

```
var edad= 22

if(edad > 18){
    println("Es mayor de edad")
}
```

Condiciones con ***de otro modo***

```
var edad= 22

if(edad > 18){
    println("Es mayor de edad")
}else{
    println("Es menor de edad")
}
```

Condiciones anidadas

```
var animal= "rooster"

if(animal == "dog"){
    println("Es un perro")
}else if(animal == "cat"){
    println("Es un gato")
}else if(animal == "bird"){
    println("Es un pájaro")
}else{
    println("Es otro animal")
}
```

Las condiciones en Kotlin también brinda la posibilidad de devolver un resultado tal como se hace con las funciones. Un ejemplo es lo siguiente:

```
val deseo = "Casa"
var compra = if( deseo == "Casa" ){
    "Carro"
}else{
    "Casa"
}
```

En el anterior código la variable compra quedaría almacenado el valor “Carro”

**Condiciones con when**

Las casos (en Java *switch*) cambian en sintaxis y palabras reservadas con respecto a Java. En Kotlin se debe usar la palabra *when.*

```
val mes = 7
```

```
when (mes) {
    1 -> print("Enero")
    2 -> print("Febrero")
    3 -> print("Marzo")
    4 -> print("Abril")
    5 -> print("Mayo")
    6 -> print("Junio")
    7 -> print("Julio")
    8 -> print("Agosto")
    9 -> print("Septiembre")
    10 -> print("Octubre")
    11 -> print("Noviembre")
    12 -> print("Diciembre")
    else -> {
        print("No corresponde a ningún mes del año")
    }
}
```

*when* es una opción mucho más potente que su equivalente en Java con *switch*. La siguiente sintaxis es válida en Kotlin:

```
val mes = 7

when (mes) {
    1,2,3 -> print("Primer trimestre del año")
    4,5,6 -> print("segundo trimestre del año")
    7,8,9 -> print("tercer trimestre del año")
    10,11,12 -> print("cuarto trimestre del año")
}
```

Otra opción es usar *in* dentro de *when*:

```
val mes = 7

when (mes) {
    in 1..6 -> print("Primer semestre")
    in 7..12 -> print("segundo semestre")
    !in 1..12 -> print("no es un mes válido")
}
```

También se puede hacer una comparación de tipos de datos usando *is*:

```
val valor:Any = 12

when (valor){
    is Int -> print(valor + 1)
    is String -> print("El texto es $valor")
```

```
    is Boolean -> if (valor) print("es verdadero") else print("es falso")
}
```

Incluso se puede usar la expresión *when* como función y almacenar la respuesta de esta:

```
val mes = 7

val respuesta : String = when (mes) {
    in 1..6 -> "Primer semestre"
    in 7..12 -> "segundo semestre"
    !in 1..12 -> "no es un mes válido"
    else ->  "error"
}
```

**Funciones**

Para la declaración de funciones sin retorno se debe usar la palabra reservada *fun* seguida del identificador de la función y los paréntesis con los parámetros (si hay varios parámetros se separan con coma):

```
private fun mostrarInformacion( mensaje : String ){
    println("Mostrar informacion $mensaje")
}
```

Para los casos en que es necesario el retorno de un valor se debe usar la siguiente sintaxis:

```
private fun dividir(numero1:Long, numero2:Long):Float{
    return (numero1/numero2).toFloat()
}
```

Las funciones en Kotlin permite el uso de argumentos (parámetros) por defecto y el la reducción de código eliminado las llaves, un ejemplo es los siguiente:

```
fun concatenarNombre(nombre:String="N/A", apellido:String="N/A"): String =
"$nombre $apellido"
```

La anterior función permite ser llamada sin necesidad del envío de argumentos, o solo mandando uno.

```
concatenarNombre("Ash")
```

El resultado de la función sería:

“Ash N/A”

O para concatenar el retorno de una función

```
println("El nombre es: ${concatenarNombre("Carlos")}")
```

Para la sobre escritura de una función se debe usar la palabra reservada *override*:

```
override fun toString(): String {
    return super.toString()
}
```

Otras posibilidades con las funciones en Kotlin:

- Recibir como parámetro una función.
- Retornar una función.
- Agregar una función dentro de otra.
- Funciones anónimas.
- Mucho más.

**Modificadores de visibilidad**

Cuando se programa con Kotlin usando el paradigma orientado a objetos se usan los siguiente modificadores de visibilidad:

- *private*: solo es visible dentro de las clase.
- *protected*: igual que la private, más visibilidad en las subclases.
- *internal*: es visible en todo el modulo al que pertenece.
- *public*: valor por defecto. Es visible para todo el proyecto.

**Ciclos**

El clásico ciclo *for* es representado en Kotlin de forma similar al *foreach* de Java.

```
for ( numero in 1..100 ){
    println("el valor es: $numero")
}
```

Para recorridos inversos se puede usar la opción *downTo.*

```
for ( numero in 10 downTo 1 ){
    println("el valor es: $numero")
}
```

Para los incrementos del índice en el valor que se desee se usa *step*.

```
for ( numero in 1..10 step 2 ){
    println("el valor es: $numero")
}
```

Existe también la posibilidad de crear un rango en el cual se evita el último elemento. Para esto se debe usar *until*.

```
for ( numero in 1 until 4){
    print("$numero,")
}
```

El resultado del ciclo anterior sería: 1,2,3,

El ciclo *while* y el *do-while* son equivalentes en Java,

```
var i = 0

while( i < 10 ){
    println("val: $i")
    i++
}

// o

do{
    println("val: $i")
    i--
}while( i > 0 )
```

*break* y *continue* también son equivalentes al uso en Java.

**Arreglos**

Los arreglos en Kotlin al igual que en Java no pueden cambiar de tamaño. Su declaración e inicialización es la siguiente:

```
val dias:Array<String>
dias= arrayOf("Lunes","Martes","Miercoles","Jueves","Sabado","Domingo")
```

La primera posición del arreglo es la cero, y la última es la del tamaño del arreglo menos uno. Por lo cual para obtener la primero y última posición del anterior arreglo, se debe hacer lo siguiente:

```
dias[0]
dias.get(0)
// y
dias[dias.size-1]
dias.get(dias.count()-1)
```

Como se puede ver se pueden usar los clásicos corchetes o el método *get()* y para obtener el tamaño del arreglo se puede usar *size* o *count().*

Para modificar el contenido de un elemento del arreglo se puede hacer los siguiente:

```
dias[0] = "Gran lunes"
dias.set(3, "Increíble viernes")
```

El recorrido de los arreglos es similar al que se realiza con los *foreach*, *while* y *do-while* en Java. El siguiente código permite ver el contenido del arreglo:

```
for ( dia in dias ){
    println("día: $dia")
}
```

Y si se desea recorrer los índices:

```
for ( indice in dias.indices ){
    println("indice: $indice")
}
```

Pero si se necesita tanto el elemento como el índice:

```
for ( (indice, dia) in dias.withIndex() ){
    println("indice: $indice y día: $dia")
}
```

Por último para crear una arreglo vacío se usa lo siguiente:

```
val arreglo = arrayOfNulls<Int>(100)
```

**Listas**

Las listas en Kotlin se pueden clasificar en no mutables (solo lectura) y mutables. Para las no mutables se puede usar la siguiente sintaxis en sus operaciones más importantes:

Declaración e inicialización

```
val soloLectura: List<Int> = listOf(1,2,3,4,5,6,7,8,9,0)
```

Operaciones

```
soloLectura.size //Muestra el tamaño de la lista
soloLectura.get(3) //Devuelve el valor de la posición 3
soloLectura.first() //Devuelve el primer valor
soloLectura.last() //Devuelve el último valor
println(soloLectura) //[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
```

En cuanto al recorrido, se usa la misma sintaxis que con los arreglos. La ventaja de usar las listas no mutables es que se maneja de forma más eficiente la memoria.

Las listas no mutables tienen las mismas funcionalidades que las de solo lectura, y además de eso la capacidad de cambiar los valores de los elementos, eliminarlos y agregar elementos nuevos.

Declaración con valores

```
val listaMutable: MutableList<Int> = mutableListOf(1,2,3,4,5,6,7,8,9)
```

O sin valores

```
val meses:MutableList<String> = mutableListOf()
```

Para agregar un elemento se puede agregándolo usando *add* (lo pone al final de la lista)

```
meses.add("Enero")
```

O indicando la posición donde lo desea almacenar (Similar a las listas en Java).

```
meses.add(0,"Meses")
```

Las opciones básicas para eliminar elementos desde la lista mutable se puede usar una posición o un elemento.

```
meses.removeAt(0)
meses.remove("Febrero")
```

La primera opción elimina la posición indicada y devuelve como resultado el elemento eliminado. La segunda opción elimina al elemento que se especifique, de encontrarlo en la lista lo elimina y retorna true, de lo contrario devuelve false.

Otras opciones interesantes con las listas mutables son:

```
meses.none() //Nos devuelve un true si la lista está vacía
meses.firstOrNull() //devuelve el primer campo, y si no hay elemento devuelve
un null.
meses.elementAtOrNull(2) //El elemento del índice 2, si no hay, devolverá un
null
meses.lastOrNull() //Último valor de la lista o null
```

Para el recorrido de las listas se puede usar la misma sintaxis que con los arreglos, entre estas opciones está el *forEach* para Kotlin.

```
meses.forEach { println(it) }
```

Como se ve por medio del índice it se puede acceder a cada una de los elementos de la lista (o arreglo).

Además de las anteriores hay una importante variedad de colecciones que permiten bajar la complejidad de los algoritmos.

**Clases**

La sintaxis para programar una clase con Kotlin es igualmente sencilla.

```
class Persona{

}
```

Para el método constructor se debe usar la palabra reservada *constructor* en la misma línea de la declaración de la clase. Cuando el constructor esta al iniciar la clase, se llama constructor primario.

```
class Persona constructor(nombre: String){

}
```

Si el constructor no tiene ninguna anotación o modificadores de visibilidad, la palabra clave constructor se puede omitir:

```
class Persona (nombre: String){

}
```

También se puede contar con un constructor similar al clásico que se usa en Java, a estos se les conoce como constructores secundarios.

```
class Person {
    constructor(parent: Person) {
        //...
    }
}
```

Si la clase tiene un constructor primario, cada constructor secundario necesita delegar en el constructor primario, ya sea directa o indirectamente a través del constructor secundario. La delegación a otro constructor de la misma clase se hace usando la palabra clave this:

```
class Person(val nombre: String) {
    constructor(nombre : String, edad: Int): this(nombre) {

    }
}
```

Tenga en cuenta que el código en los bloques de inicializador se convierte efectivamente en parte del constructor primario. La delegación en el constructor primario ocurre como la primera declaración de un constructor secundario, por lo que el código en todos los bloques de inicializador se ejecuta antes que el cuerpo constructor secundario. Incluso si la clase no tiene un constructor primario, la delegación sigue ocurriendo implícitamente, y los bloques del inicializador aún se ejecutan. Por los cual el lugar adecuado para la inicialización de los parámetros es el bloque *init*.

```
class Persona(val nombre: String) {

    init {
        // bloque de incialización
    }

    constructor(nombre : String, edad: Int) : this(nombre) {

    }
}
```

Si una clase (no abstracta) no declara ningún constructor (primario o secundario), tendrá un constructor primario generado sin argumentos. La visibilidad del constructor será pública. Si no

desea que su clase tenga un constructor público, debe declarar un constructor primario vacío con visibilidad privada.

```
class Persona private constructor(){

}
```

Por defecto en Kotlin todas las clases son *final*, por lo cual si se quiere dejar que una clase se pueda heredar, esta debe estar acompañada del modificador *open*.

```
open class Persona(val nombre: String) {

}

class Empleado(nombre:String): Persona(nombre) {

}
```

Al igual que Java no se permite herencia múltiple.

Para más información: https://kotlinlang.org/docs/reference/classes.html

**Interfaces**

Las interfaces en Kotlin son muy similares a Java 8. Pueden contener declaraciones de métodos abstractos, así como implementaciones de métodos.

```
interface OperacionesMatematicas{

    fun sumar( n1:Int, n2:Int ):Int
    fun restar( n1:Int, n2:Int ):Int

    fun multiplicar( n1:Int, n2:Int ):Int{
        return n1 * n2;
    }

    fun dividir(n1:Int, n2:Int ):Int
}
```

Para heredar de una interfaz desde una clase se usa la misma sintaxis que la herencia entre clase, y al igual que en Java se permite la herencia múltiple de interfaces.

**Estándares de codificación y documentación**

Para el desarrollo de aplicaciones usando Kotlin se usarán las mismas buenas prácticas que se tienen con Java.

## Para realizar en clase

**1)** Escribir en kotlin una función recursiva que divida dos números enteros por medio de restas sucesivas. Ejemplo: 12/3 es lo mismo que decir: 12 - 3 = 9, 9 - 3 = 6, 6 - 3 = 3, 3 - 3 = 0. Hubo 4 restas hasta que dio 0, por lo tanto 12/3 = 4.

**2)** Se debe diseñar el diagrama de clases y el código en Kotlin que supla las necesidades planteadas en el siguiente enunciado.

Un software necesita gestionar información de los empleados y clientes de una empresa. Donde los empleados y clientes poseen datos como el nombre completo, documento de identidad y correo electrónico.

Además, cada empleado tiene asignado un salario único, dependencia (ventas, recursos humanos, gerencia, operativo), y puede tener un conjunto de empleados subordinados.

De los clientes además se necesita conocer su dirección de correspondencia y el teléfono.

Un empleado siempre tendrá asignado un cargo, el cual tiene un nombre y un nivel jerárquico dentro de la empresa.

La empresa tiene una razón social (nombre), un NIT y una dirección. Y podrá tener o no clientes asociados, pero siempre necesitará empleados.

El software necesita las operaciones CRUD tanto para clientes como para empleados, además se desea obtener el valor de la nómina de toda la empresa y la nómina por cada dependencia. Tenga en cuenta que no se requiere interfaz gráfica de usuario.