Aprende el lenguaje de programación Kotlin

Kotlin es un lenguaje de programación ampliamente utilizado por los desarrolladores de Android de todo el mundo. Este tema funciona como un curso intensivo de Kotlin para que puedas configurarlo y comenzar a utilizarlo rápidamente.

1. Variable

1.1 Declaración de variable

Kotlin utiliza dos palabras clave diferentes para declarar variables: `val` y `var`.

- Usa `val` para una variable cuyo valor no cambia nunca. No puedes volver a asignar un valor a una variable que se declaró mediante `val`.
- Utiliza `var` para una variable cuyo valor puede cambiar.

En el siguiente ejemplo, `count` es una variable de tipo `Int` asignada a un valor inicial de `10`:

```
var count: Int = 10
```

`Int` es un tipo que representa un número entero, uno de los muchos tipos numéricos que se pueden representar en Kotlin. Del mismo modo que con otros lenguajes, también puedes utilizar `Byte`, `Short`, `Long`, `Float` y `Double` según tus datos numéricos.

La palabra clave `var` quiere decir que puedes volver a asignar valores a `count` según sea necesario. Por ejemplo, puedes cambiar el valor de `count` de `10` a `15`:

```
var count: Int = 10
count = 15
```

Sin embargo, algunos valores no se pueden cambiar. Tomemos como ejemplo un objeto `String` llamado `languageName`. Si quieres asegurarte de que `languageName` siempre retenga un valor de "Kotlin", puedes declarar `languageName` mediante la palabra clave `val`:

```
val languageName: String = "Kotlin"
```

Estas palabras clave te permiten ser explícito acerca de lo que se puede cambiar. Úsalas como más te convenga. Si es necesario que la referencia de una variable se pueda volver a asignar, declárala como `var`. De lo contrario, usa `val`.

1.2 Inferencia de tipo

Continuando con el ejemplo anterior, cuando asignas un valor inicial a `languageName`, el compilador de Kotlin puede inferir el tipo en función del tipo del valor asignado.

Debido a que el valor de `"Kotlin"` es de tipo `String`, el compilador infiere que `languageName` también es `String`. Ten en cuenta que Kotlin es un lenguaje de *tipo estático*. Eso quiere decir que el tipo se resuelve en el tiempo de compilación y no cambia nunca.

En el siguiente ejemplo, `languageName` se infiere como `String`, de manera que no puedes llamar a ninguna de las funciones que no forman parte de la clase de `String`:

```
val languageName = "Kotlin"
val upperCaseName = languageName.toUpperCase()

// Fails to compile
languageName.inc()
```

`toUpperCase()` es una función a la que solo se puede llamar en variables de tipo `String`. Debido a que el compilador de Kotlin infirió `languageName` como `String`, puedes llamar de manera segura a `toUpperCase()`. `inc()`, sin embargo, es una función de operador de `Int`, de manera que no se la puede llamar en un objeto `String`. El enfoque de Kotlin para las inferencias de tipo garantiza la concisión y ofrece seguridad de tipo.

1.3 Seguridad nula

En algunos ejemplos, se puede declarar una variable de tipo de referencia sin proporcionar un valor explícito inicial. En estos casos, por lo general, la variable contiene un valor nulo. Las variables de Kotlin no pueden retener valores nulos de manera predeterminada. Por lo tanto, el siguiente fragmento no es válido:

```
// Fails to compile
val languageName: String = null
```

Para que una variable retenga un valor nulo, debe ser de tipo *anulable*. Si deseas especificar que una variable es anulable, puedes agregar un sufijo a su tipo mediante `?`, como se muestra en el siguiente ejemplo:

```
val languageName: String? = null
```

Mediante el tipo `String?`, puedes asignar un valor `String` o `null` a `languageName`.

Debes manejar las variables anulables con cuidado o podrías obtener un `NullPointerException` no deseado. En Java, por ejemplo, tu programa falla si intentas invocar un método en una variable nula.

Kotlin ofrece mecanismos para trabajar de manera segura con variables anulables. Si deseas obtener más información, consulta Patrones comunes en Android: Nulabilidad.

2. Condicionales

Kotlin cuenta con varios mecanismos para implementar la lógica condicional. El más común es la *declaración "if-else"*. Si una expresión entre paréntesis junto a una palabra clave `if` evalúa `true`, se ejecuta el código dentro de esa rama (es decir, el código que siga inmediatamente después del código entre paréntesis). De lo contrario, se ejecuta el código dentro de la rama de `else`.

```
if (count == 42) {
    println("I have the answer.")
} else {
    println("The answer eludes me.")
}
```

Puedes representar varias condiciones mediante `else if`. De esta manera, puedes representar una lógica más detallada y compleja dentro de un solo enunciado condicional, como se muestra en el siguiente ejemplo:

```
if (count == 42) {
    println("I have the answer.")
} else if (count > 35) {
    println("The answer is close.")
} else {
    println("The answer eludes me.")
}
```

Las declaraciones condicionales son útiles para representar lógica de estado, aunque podría resultarte repetitivo escribirlas. En el ejemplo anterior, imprimes un objeto `String` en cada rama. Para evitar esta repetición, Kotlin ofrece *expresiones condicionales*. El último ejemplo se puede volver a escribir de la siguiente manera:

```
val answerString: String = if (count == 42) {
    "I have the answer."
} else if (count > 35) {
    "The answer is close."
} else {
    "The answer eludes me."
}

println(answerString)
```

De manera implícita, cada rama condicional muestra el resultado de la expresión en su línea final. Por lo tanto, no necesitas usar una palabra clave `return`. Como el resultado de las tres ramas es de tipo `String`, el resultado de la expresión "if-else" también es de tipo `String`. En este ejemplo, a `answerString` se le asigna un valor inicial del resultado de la expresión "if-else". La inferencia de tipo se puede utilizar a fin de omitir la declaración explícita de tipo para `answerString`, aunque suele convenir incluirla para mayor claridad.

Nota: Kotlin no incluye un operador ternario, sino que favorece el uso de expresiones condicionales.

A medida que aumenta la complejidad de tu declaración condicional, quizás te convenga reemplazar la expresión "if-else" con una expresión *when*, como se muestra en el siguiente ejemplo:

```
val answerString = when {
    count == 42 -> "I have the answer."
    count > 35 -> "The answer is close."
    else -> "The answer eludes me."
}

println(answerString)
```

Cada rama en una expresión `when` se representa mediante una condición, una flecha (`->`) y un resultado. Si la condición en el lado izquierdo de la flecha se evalúa como verdadera, entonces se muestra el resultado de la expresión en el lado derecho. Ten en cuenta que la ejecución no pasa de una rama a la otra. El código en la expresión de ejemplo de `when` cuenta con las mismas funciones que el código en el ejemplo anterior, pero es más fácil de leer.

Los condicionales de Kotlin destacan una de sus funciones más potentes: la *conversión inteligente*. En lugar de usar el operador de llamada segura o el de aserción que no es nulo para trabajar con valores anulables, puedes verificar si una variable contiene una referencia a un valor nulo mediante una declaración condicional, como se muestra en el siguiente ejemplo:

```
val languageName: String? = null
if (languageName != null) {
    // No need to write languageName?.toUpperCase()
    println(languageName.toUpperCase())
}
```

Dentro de la rama condicional, se trata a `languageName` como no anulable. Kotlin es lo suficientemente inteligente para reconocer que la condición para ejecutar la rama es que `languageName` no tenga un valor nulo. Por lo tanto, no es necesario que trates a `languageName` como anulable dentro de esa rama. Esta conversión inteligente funciona para las verificaciones nulas, las verificaciones de tipo o cualquier otra condición que cumpla un contrato.

3. Bucles

Los bucles son estructuras de control básicas para ejecutar fragmentos de código múltiples veces de forma consecutiva sin tener que escribir el mismo código repetido múltiples veces, en Kotlin podemos usar los bucles for, while y do while.

3.1 Bucles for en Kotlin

El bucle for es el que se utiliza más habitualmente y en Kotlin es un poco distinto a como es en Java, aquí tenemos una variable que itera sobre un rango o sobre una lista, array, map,… y como tal en cada iteración toma el siguiente valor.

Por ejemplo para imprimir los números del 1 al 5 podemos hacer un bucle for en el que indiquemos el rango 1..5 y no tenemos que preocuparnos de declarar el índice ni de actualizarlo.

```
for(num in 1..5) {
    println("numero: $num")
}
/* Resultado:
numero: 1
numero: 2
numero: 3
numero: 4
numero: 5
*/
```

Así en Kotlin podemos usar bucles for con la misma estructura ya sea para recorrer algún tipo de colección de objetos o para ejecutarlo un número determinado de veces (un rango).

3.2 Bucles while en Kotlin

Con while podemos hacer que se ejecute un bloque de código determinado hasta que se cumpla una condición.

Cuando usamos este tipo de bucles hay que tener cuidado para evitar producir bucles infinitos no deseados, para lo que necesitamos que dentro del bucle se actualice el valor que usemos en la condición de salida bajo alguna circunstancia.

En el siguiente ejemplo se ejecuta el bucle mientras (while) el día sea menor que 6 y cuando deja de cumplirse termina.

```
var dia= 1
println("Empiza la semana")
while(dia < 6) {
    if  (dia == 1) {
        println("$dia dia trabajando")
    } else {
        println("$dia dias trabajando")
    }
    dia++ // Actualizamos la condicion
}
println("A descansar")
/* Resultado:
Empiza la semana
1 dia trabajando
2 dias trabajando
3 dias trabajando
4 dias trabajando
5 dias trabajando
A descansar
*/
```

3.3 Bucles do while en Kotlin

Los bucles do while son muy similares a los bucles while, puesto que en ambos se evalúa una condición y mientras se siga cumpliendo se seguirá ejecutando. La diferencia reside en que con while la condición se evalúa antes de entrar al bucle por lo que existe la posibilidad de que no se entre nunca si no se cumple la condición, mientras que con el do while la comprobación se hace después de cada iteración asique al menos se va a ejecutar una vez.

Por ejemplo si queremos que el usuario introduzca un número dentro de un rango el do while es una buena opción porque se ejecutará al menos una vez y si mete un número no válido pues seguimos en el bucle hasta que se decida a poner uno correcto.

```
var numero:Int
do {
    println("Introduce un numero entre 1 y 100")
    numero = readLine()!!.toInt()
} while(numero !in 1..100) // numero < 1 || numero > 100
println("Gracias")
/* Resultado:
Introduce un numero entre 1 y 100
423
Introduce un numero entre 1 y 100
-71
Introduce un numero entre 1 y 100
7
Gracias
*/
```

Como salir de un bucle en Kotlin (break y continue)

A veces queremos que un bucle deje de ejecutarse antes de que termine si se da alguna condición o también puede ser que queramos que no se ejecute el contenido del bucle en algunos casos pero sí que queremos que el bucle

Saltar a la siguiente iteración con continue

Con continue podemos hacer que se deje de ejecutarse el código dentro del bucle y que se pase directamente a la siguiente iteración, por ejemplo, si queremos hacer en Kotlin un bucle que solo imprima los números impares podríamos hacerlo continue de la siguiente forma:

```
for (num in 1..10) {
    if (num % 2 == 0) {
        continue
    }
    print("$num ")
}
/* Resultado:
1 3 5 7 9
*/
```

Como el flujo de ejecución se interrumpe al llegar al continue solo se ejecuta el print("$num ") cuando el número no es par.

Salir de un bucle con break

Mientras que con continue seguimos dentro del bucle con break se sale completamente aunque aún no se haya cumplido la condición para que termine. Vamos con otro ejemplo con números, supongamos que queremos que se impriman todos los números de un rango, pero si encontramos un múltiplo de 5 el bucle tiene que terminarse.

```
for (num in 1..10) {
    if (num % 5 == 0) {
        break
    }
    print("$num ")
}
/* Resultado:
1 2 3 4
*/
```

Cuando se ejecuta un break dentro de un bucle esto implica que se sale inmediatamente.

Salir de un bucle con break desde un bucle anidado

El break provoca que termine el bucle en el que se encuentra, pero es posible que en realidad queramos que termine un bucle superior. Para esto tenemos que utilizar además del break una etiqueta para indicar a qué bucle debe de afectar el break, la forma de hacer esto es añadiendo @nombre al break (sin dejar espacios entre medias) y nombre@ antes del bucle al que afectará.

Y para terminar un ejemplo sencillo que imprime todos los múltiplos de cada uno de los números del bucle exterior hasta que llega a los múltiplos de 5.

```
bublePrincipal@ for(num in 1..10) {
    println("Multiplos de $num: ")
    for (num2 in 1..10) {
        if (num2 % num == 0)  {
            if (num % 5 == 0) {
                println("Ya me canse")
                break@bublePrincipal
            }
            print("$num2 ")
        }
    }
    println()
}
/* Resultado:
Multiplos de 1:
1 2 3 4 5 6 7 8 9 10
Multiplos de 2:
2 4 6 8 10
Multiplos de 3:
3 6 9
Multiplos de 4:
4 8
Multiplos de 5:
Ya me canse
*/
```

4. Funciones

Puedes agrupar una o más expresiones en una *función*. En lugar de repetir la misma serie de expresiones cada vez que necesitas un resultado, puedes unir las expresiones en una función y llamar a esa función.

Para declarar una función, usa la palabra clave de `fun` seguida de un nombre de función. A continuación, define los tipos de entrada que tu función acepta, si corresponde, y declara el tipo de resultado que muestra. El cuerpo de una función es el lugar en el que defines las expresiones a las que se llama cuando se invoca tu función.

En función de los ejemplos anteriores, la siguiente es una función de Kotlin completa:

```
fun generateAnswerString(): String {
    val answerString = if (count == 42) {
        "I have the answer."
    } else {
        "The answer eludes me"
    }

    return answerString
}
```

La función en el ejemplo anterior tiene el nombre `generateAnswerString`. No requiere ninguna entrada. Produce un resultado de tipo `String`. Para llamar a una función, usa su nombre seguido del operador de invocación (`()`). En el siguiente ejemplo, la variable de `answerString` se inicializa con el resultado desde `generateAnswerString()`.

```
val answerString = generateAnswerString()
```

Las funciones pueden tomar argumentos como entrada, como se muestra en el siguiente ejemplo:

```
fun generateAnswerString(countThreshold: Int): String {
    val answerString = if (count > countThreshold) {
        "I have the answer."
    } else {
```

```
        "The answer eludes me."
    }

    return answerString
}
```

Cuando declaras una función, puedes especificar la cantidad de argumentos que desees y sus tipos. En el ejemplo anterior, `generateAnswerString()` toma un argumento con el nombre `countThreshold` de tipo `Int`. Dentro de la función, puedes usar el nombre del argumento para hacer referencia a él.

Cuando llamas a esta función, debes incluir un argumento dentro del paréntesis de la llamada a la función:

```
val answerString = generateAnswerString(42)
```

4.1 Cómo simplificar declaraciones de funciones

`generateAnswerString()` es una función bastante sencilla. La función declara una variable y se muestra inmediatamente después. Cuando el resultado de una sola expresión se muestra desde una función, puedes dejar de declarar una variable local. Para ello, muestra directamente el resultado de la expresión "if-else" incluido en la función, como se muestra en el siguiente ejemplo:

```
fun generateAnswerString(countThreshold: Int): String {
    return if (count > countThreshold) {
        "I have the answer."
    } else {
        "The answer eludes me."
    }
}
```

También puedes reemplazar la palabra clave que se mostró con el operador de asignación:

```
fun generateAnswerString(countThreshold: Int): String = if (count >
countThreshold) {
        "I have the answer"
    } else {
        "The answer eludes me"  }
```

4.2 Funciones anónimas

No todas las funciones necesitan un nombre. Algunas funciones se pueden identificar de manera más directa a través de sus entradas y resultados. Estas funciones se conocen como *funciones anónimas*. Puedes mantener una referencia a una función anónima y utilizarla para llamar a la función más tarde. También puedes pasar la referencia a tu aplicación, del mismo modo que lo haces con otros tipos de referencia.

```
val stringLengthFunc: (String) -> Int = { input ->
    input.length
}
```

Al igual que las funciones con nombre, las funciones anónimas pueden contener cualquier cantidad de expresiones. El valor que se muestra de la función es el resultado de la expresión final.

En el ejemplo anterior, `stringLengthFunc` contiene una referencia a una función anónima que toma un objeto `String` como entrada y muestra la longitud del objeto `String` de entrada como resultado del tipo `Int`. Por ese motivo, el tipo de la función se denota como `(String) -> Int`. Sin embargo, este código no invoca la función. Si quieres recuperar el resultado de la función, debes invocarlo del mismo modo que lo harías con una función con nombre. Debes proporcionar un objeto `String` cuando llamas a `stringLengthFunc`, como se muestra en el siguiente ejemplo:

```
val stringLengthFunc: (String) -> Int = { input ->
    input.length
}

val stringLength: Int = stringLengthFunc("Android")
```

4.3 Funciones de orden superior

Una función puede tomar otra función como argumento. Las funciones que usan otras funciones como argumentos se llaman *funciones de orden superior*. Este patrón es útil para la comunicación entre los componentes. Del mismo modo, puedes usar una interfaz de devolución de llamada en Java.

El siguiente es un ejemplo de una función de orden superior:

```
fun stringMapper(str: String, mapper: (String) -> Int): Int {
    // Invoke function
    return mapper(str)
}
```

La función `stringMapper()` toma un objeto `String` además de una función que deriva en un valor de `Int` a partir de un objeto `String` que pasas a la función.

Para llamar a `stringMapper()`, puedes pasar un objeto `String` y una función que satisfaga el otro parámetro de entrada, es decir, una función que tome un objeto `String` como entrada y muestre un objeto `Int`, como se indica en el siguiente ejemplo:

```
stringMapper("Android", { input ->
    input.length
})
```

Si la función anónima es el *último* parámetro que se definió en una función, puedes pasarla fuera de los paréntesis que se utilizaron para invocar la función, como se muestra en el siguiente ejemplo:

```
stringMapper("Android") { input ->
    input.length
}
```

Las funciones anónimas se pueden encontrar en toda la biblioteca estándar de Kotlin. Si deseas obtener más información, consulta Lambdas y funciones de orden superior.

5. Clases

Todos los tipos mencionados hasta ahora están integrados en el lenguaje de programación Kotlin. Si deseas agregar tu propio tipo personalizado, puedes definir una clase mediante la palabra clave `class`, como se muestra en el siguiente ejemplo:

```
class Car
```

5.1 Propiedades

Las clases representan propiedades que usan estados. Una propiedad es una variable a nivel de la clase que puede incluir un método get, un método set y un campo de copia de seguridad. Ya que un auto necesita ruedas para andar, puedes agregar una lista de objetos `Wheel` como una propiedad de `Car`, como se muestra en el siguiente ejemplo:

```
class Car {
    val wheels = listOf<Wheel>()
}
```

Ten en cuenta que `wheels` es un `public val`, lo cual quiere decir que se puede acceder a `wheels` desde fuera de la clase `Car` y que no se puede volver a asignar. Si deseas obtener una instancia de `Car`, primero debes llamar a su constructor. Desde allí, puedes acceder a cualquiera de sus propiedades de accesibilidad.

```
val car = Car() // construct a Car
val wheels = car.wheels // retrieve the wheels value from the Car
```

Si quieres personalizar tus ruedas, puedes definir un constructor personalizado que especifique cómo se inicializan las propiedades de tu clase.

```
class Car(val wheels: List<Wheel>)
```

En el ejemplo anterior, el constructor de clase toma un objeto `List<Wheel>` como argumento de constructor y usa ese argumento para inicializar su propiedad de `wheels`.

5.2 Encapsulación y funciones de clase

Las clases usan funciones para modelar el comportamiento. Las funciones pueden modificar el estado, lo que te ayuda a exponer solamente los datos que deseas exponer. Este control de acceso forma parte de un concepto orientado a objetos de mayor tamaño, conocido como *encapsulación*.

En el siguiente ejemplo, la propiedad de `doorLock` se mantiene en privado respecto de cualquier elemento fuera de la clase de `Car`. Para desbloquear el auto, debes llamar a la función `unlockDoor()` y pasar una clave válida, como se muestra en el siguiente ejemplo:

```
class Car(val wheels: List<Wheel>) {

    private val doorLock: DoorLock = ...

    fun unlockDoor(key: Key): Boolean {
        // Return true if key is valid for door lock, false otherwise
    }
}
```

Si deseas personalizar la manera en la que se hace referencia a una propiedad, puedes proporcionar un método get y un método set personalizados. Por ejemplo, si quieres exponer el método get de una propiedad al mismo tiempo que restringes el acceso a su método set, puedes designar ese método set como `private`:

```
class Car(val wheels: List<Wheel>) {

    private val doorLock: DoorLock = ...

    var gallonsOfFuelInTank: Int = 15
        private set

    fun unlockDoor(key: Key): Boolean {
        // Return true if key is valid for door lock, false otherwise
    }
}
```

Con una combinación de propiedades y funciones, puedes crear clases que modelen todo tipo de objetos.

6. Interoperabilidad

Una de las funciones más importantes de Kotlin es la excelente interoperabilidad que tiene con Java. Debido a que el código Kotlin compila el código de bytes de JVM, tu código Kotlin puede llamar directamente al código Java, y viceversa. Eso quiere decir que puedes aprovechar las bibliotecas de Java existentes directamente desde Kotlin. Además, la mayoría de las API de Android están escritas en Java y puedes llamarlas directamente desde Kotlin.

Extra

Kotlin es un lenguaje flexible y práctico que cada vez tiene más impulso y compatibilidad. Te recomendamos que lo pruebes si todavía no lo hiciste. Para conocer los próximos pasos, consulta en la documentación oficial de Kotlin junto con la guía sobre cómo aplicar patrones comunes de Kotlin en tus apps para Android.

https://kotlinlang.org/

https://developer.android.com/kotlin/common-patterns?hl=es-419

https://kotlinlang.org/docs/kotlin-pdf.html

Android developers: https://www.youtube.com/user/androiddevelopers