LENGUAJE DE PROGRAMACIÓN PARA APLICACIONES MÓVILES

Aplicaciones Moviles

21 DE MAYO DE 2021

ALUMNA: LIZBETH VERONICA CASAS PEREZ 8º SEMESTRE GRUPO “A”
DOCENTE: RAMÍREZ SANTIAGO BENEDICTO

ELEMENTOS BASICOS PARA APLICACIONES MOVILES:

KOTLIN BASICO:

Es un lenguaje de programación muy utilizado por desarrolladores de Android de todo el mundo

Declaración 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.

- 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.

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.")
}
```

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.

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

    return answerString
}
```

Cómo simplificar declaraciones de funciones:

generateAnswerString() 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

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

Funciones anónimas: se puede mantener una referencia a una función anónima y utilizarla para llamar a la función más tarde. También puede pasar la referencia a la aplicación, del mismo modo que lo haces con otros tipos de referencia.

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

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

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.

```
class Car
```

Propiedades

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.

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

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.

```
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
    }
}
```

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.

Condiciones compuestas con operadores lógicos

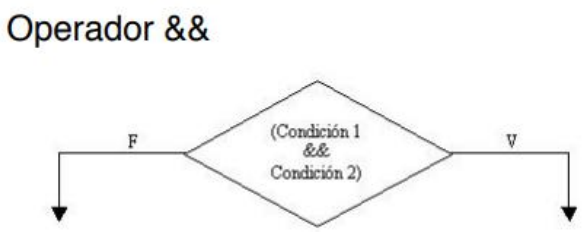
Operadores:

```
relacionales (>, <, >=, <= , ==, !=)
matemáticos (+, -, *, /, %)
```

operadores imprescindibles:

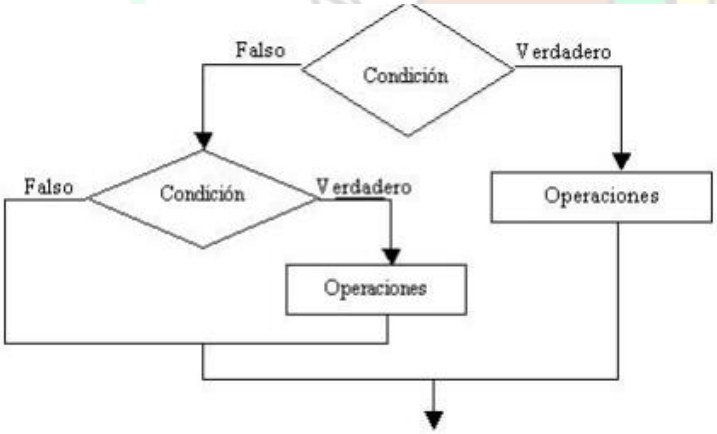
```
lógicos (&&, ||)
```

Estos dos operadores se emplean fundamentalmente en las estructuras condicionales para agrupar varias condiciones simples.



Estructuras condicionales anidadas

Decimos que una estructura condicional es anidada cuando por la rama del verdadero o el falso de una estructura condicional hay otra estructura condicional.




```
fun main(parametros: Array<String>) {
    print("Ingrese primer nota:")
    val nota1 = readLine()!!.toInt()
    print("Ingrese segunda nota:")
    val nota2 = readLine()!!.toInt()
    print("Ingrese tercer nota:")
    val nota3 = readLine()!!.toInt()
    val promedio = (nota1 + nota2 + nota3) / 3
    if (promedio >= 7)
        print("Promocionado")
    else
        if (promedio >= 4)
            print("Regular")
        else
            print("Libre")
}
```

SENTENCIAS DE CONTROL:

Estructura repetitiva while

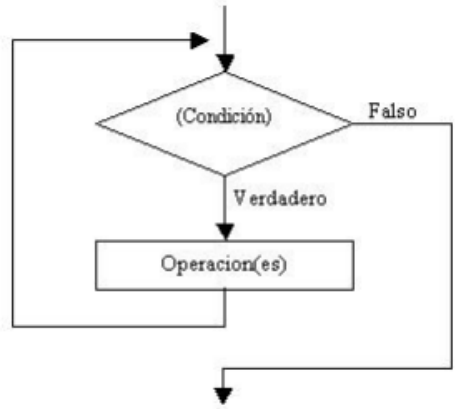
Una estructura repetitiva permite ejecutar una instrucción o un conjunto de instrucciones varias veces. Una ejecución repetitiva de sentencias se caracteriza por:

- La o las sentencias que se repiten.

- El test o prueba de condición antes de cada repetición, que motivará que se repitan o no las sentencias.

Estructura repetitiva while.

Representación gráfica de la estructura while:

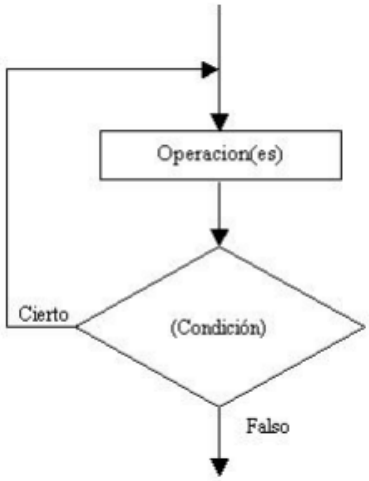


Ejemplo:

```
fun main(parametro: Array<String>) {
    var x = 1
    while (x <= 100) {
        println(x)
        x = x + 1
    }
}
```

Estructura repetitiva do/while

La estructura do/while es otra estructura repetitiva, la cual ejecuta al menos una vez su bloque repetitivo, a diferencia del while que podría no ejecutar el bloque. Esta estructura repetitiva se utiliza cuando conocemos de antemano que por lo menos una vez se ejecutará el bloque repetitivo. La condición de la estructura está abajo del bloque a repetir, a diferencia del while que está en la parte superior.




```
fun main(parametro: Array<String>) {
    do {
        print("Ingrese un valor comprendido entre 0 y 999:")
        val valor = readLine()!!.toInt()
        if (valor < 10)
            println("El valor ingresado tiene un dígito")
        else
            if (valor < 100)
                println("El valor ingresado tiene dos dígitos")
            else
                println("El valor ingresado tiene tres dígitos")
    } while (valor != 0)
}
```

Estructura repetitiva for y expresiones de rango

La estructura for tiene algunas variantes en Kotlin, en este concepto veremos la estructura for con expresiones de rango. Veamos primero como se define y crea un rango. Un rango define un intervalo que tiene un valor inicial y un valor final, se lo define utilizando el operador.

```
val docena = 1..12

if (5 in docena)
    println("el 5 está en el rango docena")

if (18 !in docena)
    println("el 18 no está en el rango docena")
```

Estructura condicional when

Además de la estructura condicional if Kotlin nos proporciona una estructura condicional para situaciones que tenemos que verificar múltiples condiciones que se resuelven con if anidados.

```
fun main(parametro: Array<String>) {
    print("Ingrese coordenada x del punto:")
    val x = readLine()!!.toInt()
    print("Ingrese coordenada y del punto:")
    val y = readLine()!!.toInt()
    when {
        x > 0 && y > 0 -> println("Primer cuadrate")
        x < 0 && y > 0 -> println("Segundo cuadrante")
        x < 0 && y < 0 -> println("Tercer cuadrante")
        x > 0 && y < 0 -> println("Cuarto cuadrante")
        else -> println("El punto se encuentra en un eje")
    }
}
```

Operadores

En una condición deben disponerse únicamente variables, valores constantes y operadores relacionales. Operadores Relacionales:

```
>  (mayor)
<  (menor)
>= (mayor o igual)
<= (menor o igual)
== (igual)
!= (distinto)
```

```
+ (más)
- (menos)
* (producto)
/ (división)
% (resto de una división)  Ej.:  x = 13 % 5  {se guarda 3}
```

LAYOUTS

Un diseño define la estructura de una interfaz de usuario en tu aplicación, por ejemplo, en una actividad. Todos los elementos del diseño se crean usando una jerarquía de objetos View y ViewGroup. Una View suele mostrar un elemento que el usuario puede ver y con el que puede interactuar. En cambio, un ViewGroup es un contenedor invisible que define la estructura de diseño de View y otros objetos:

ViewGroup:

LinearLayout: Dispone los elementos en una fila o en una columna.

TableLayout: Distribuye los elementos de forma tabular.

RelativeLayout: Dispone los elementos en relación a otro o al padre.

ConstraintLayout: Versión mejorada de RelativeLayout, que permite una edición visual desde el editor.

FrameLayout: Permite el cambio dinámico de los elementos que contiene.

AbsoluteLayout: Posiciona los elementos de forma absoluta.

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a Button" />
</LinearLayout>
```

Atributos:

Cada objeto View y ViewGroup admite su propia variedad de atributos XML. Algunos atributos son específicos de un objeto View (por ejemplo, TextView admite el atributo textSize), aunque estos atributos también son heredados por cualquier objeto View que pueda extender esta clase. Algunos son comunes para todos los objetos View, porque se heredan de la clase raíz View (como el atributo id). Además, otros atributos se consideran "parámetros de diseño", que son atributos que describen ciertas orientaciones de diseño de View, tal como lo define el objeto ViewGroup superior de ese objeto.

ID

Cualquier objeto View puede tener un ID entero asociado para identificarse de forma única dentro del árbol. Cuando se compila la aplicación, se hace referencia a este ID como un número entero, pero normalmente se asigna el ID en el archivo XML de diseño como una string del atributo id. Este es un atributo XML común para todos los objetos View (definido por la clase View) y lo utilizarás muy a menudo. La sintaxis de un ID dentro de una etiqueta XML es la siguiente:

```xml
android:id="@+id/my_button"
```

El símbolo arroba (@) al comienzo de la string indica que el analizador de XML debe analizar y expandir el resto de la string de ID, y luego identificarla como un recurso de ID. El símbolo más (+) significa que es un nuevo nombre de recurso que se debe crear y agregar a nuestros recursos (en el archivo R.java). El framework de Android ofrece otros recursos de ID

```
android:id="@android:id/empty"
```

Especificar el ancho y la altura con medidas exactas, aunque no es recomendable hacerlo con mucha frecuencia. Generalmente, se usa una de estas constantes para establecer el ancho o la altura:

- ***wrap_content*** le indica a tu vista que modifique su tamaño conforme a las dimensiones que requiere su contenido.
- ***match_parent*** le indica a tu vista que se agrande tanto como lo permita su grupo de vistas superior.

Diseños comunes

Cada subclase de la clase ViewGroup proporciona una manera única de mostrar las vistas que anidas en ella. A continuación, se muestran algunos de los tipos de diseño más comunes integrados en la plataforma Android.