PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Swift
Condicionales if y switch
Iteración con for
Clases
Herencia

Estructuras condicionales

El lenguaje de programación Swift nos permite utilizar sentencias condicionales tal y como lo ofrece cualquier otro lenguaje.

En primer lugar, vamos a ver cuál es la sintaxis de la estructura if:

> Después de la palabra reservada if, deberemos colocar una **condición**, que irá **sin paréntesis**, y a continuación, el bloque que se ejecutará en caso de que la condición sea verdadera. Los **bloques** de código se delimitarán mediante **llaves** ({}).
>
> De igual forma que en otros lenguajes, aquí también disponemos del **bloque else**, cuyo bloque se ejecutará cuando la condición sea falsa.
>
> Un ejemplo de if-else lo podemos ver en el siguiente código:

Ejemplo de if-else

```
print("Introduce un entero")
var numerostring = readLine()!
var numero = Int(numerostring)!
if numero % 2 == 0
{
      print("El numero es par")
      } else { print("El numero es impar")
}
```

Si queremos evaluar varias condiciones anidadas en un mismo if, podremos utilizar los operadores lógicos AND (representado mediante **&&**) y OR (representado mediante ||).

Swift también nos ofrece la posibilidad de usar la estructura condicional switch, a la que tendremos que indicar una variable para evaluar, y realizar acciones mediante sus valores (usando la palabra reservada case).

En este caso, tendremos que poner de forma **obligatoria la opción** default, que se ejecutará cuando no se cumplan ninguno de los casos evaluados.

Ejemplo de switch

```
print("Introduce un entero")
var numerostring = readline()!
var numero = Int(numerostring)!

switch numero
{
    case 1:
    // código
    case 2:
    // código
    case 3:
    //código
}
```

Estructuras repetitivas

Como podemos imaginar, cualquier lenguaje de programación que se precie deberá ser capaz de proporcionar estructuras repetitivas, o bucles, para poder realizar tareas iterativas con ellas.

El bucle for es el primero de los que vamos a ver en el lenguaje Swift. Este bucle iterará en un intervalo de valores, por ejemplo, de 1 a 10, recorriéndolo de uno en uno.

Para crear un bucle for, deberemos escribir la palabra reservada for y una variable iteradora, seguida de in y un intervalo, que se definirá de la forma **inicio, tres puntos, fin**.

Ejemplo de for

```
for i in 1...10
{
    print("Voy por \(i)")
}
```

También podremos recorrer fácilmente los elementos de un array mediante un **bucle for**:

Ejemplo de for recorriendo un array

```
var array_de_valores = [1, 2, 3, 4, 5, 6]
for valor in array_de_valores
{
    print("Valor vale \(valor)")
}
```

El otro tipo de bucle que nos proporcionará Swift es el bucle repeat while. Este bucle se estará **ejecutando mientras** se cumpla una cierta **condición**.

Para crear un bucle repeat while, deberemos escribir la palabra repeat y un **bloque con código** a ejecutar. Por último, se escribirá la palabra while seguida de la **condición** de parada del bucle.

A estas condiciones, podremos aplicarles los operadores AND y OR de igual forma que lo hemos podido hacer con otros lenguajes.

Ejemplo de repeat while

```
var numero:Int?

repeat {
    print("Introduce un entero")
    let numerostring = readLine()!
    numero = Int(numerostring)!
} while numero! % 2 ! = 0
```

En el ejemplo de la figura anterior, estaremos leyendo números por teclado hasta que se introduzca un número par, que hará que no se cumpla la condición y se detenga el bucle.

Cláusula for(from – through – by)

Si recordamos los for en Java, a estos podíamos **indicarles** que su **incremento** fuese en el número que nosotros necesitáramos, ¿se podrá hacer lo mismo en Swift? La respuesta es que sí, en este lenguaje, podremos utilizar la cláusula from – through – by mediante la que podremos indicar desde **dónde** queremos **empezar** a iterar, **hasta dónde** queremos llegar, y su **intervalo**.

Podría utilizarse de la forma que se indica en el siguiente código.

Ejemplo de from – through – by

```
print("Introduce un entero de inicio")
var numero1string = readLine()!
var numero1 = Int(numero1string)!

print("Introduce un entero de fin")
var numero2string = readLine()!
var numero2 = Int(numero2string)!

for num in stride(from: numero1, through: numero2, by: 2)
{
    print("Recorremos los números, y encontramos el impar: \(num)")
}
```

Métodos

En Swift, podremos declarar funciones o métodos mediante la palabra reservada func, seguida del **nombre** que queramos darle al **método**.

Después del nombre, deberemos indicar *entre paréntesis la lista de parámetros* que recibirá la función, habiendo de **indicar el** tipo de los mismos. En el caso de que no se reciban parámetros, se usarán los paréntesis vacíos.

Por último, con el operador flecha (->), indicaremos el **tipo** del valor **que devolverá**, siendo Void en caso de no devolver nada. El código del mismo irá en un **bloque entre paréntesis**.

Para devolver los valores, utilizaremos return.

Ejemplo de declaración de funciones

```
// Función que recibe dos parámetros y no devuelve nada
func funcion1(parametro1:Int, parametro2:String) -> Void
{

}

// Función que no recibe parámetros y devuelve un entero
func funcion2() -> Int
{
    return 1
}
```

Una vez que hemos declarado las funciones necesarias, deberemos poder llamarlas para poder utilizarlas.

- Cuando se trate de funciones que **devuelvan un valor**, deberemos **igualarlas a una variable o constante**,
- mientras que cuando se trate de **funciones que no devuelvan** nada, podremos invocarlas **directamente** escribiendo su nombre.

En Swift, es importarte remarcar que deberemos **escribir los nombres de los parámetros** en las **llamadas** de forma obligatoria.

Ejemplo de llamadas a funciones

```
// Llamada a la función con parámetros
var valor = "Hola"
fun1(parametro1: 3, parametro2: valor)

// Lamada a la función que devuelve un valor
var resultado = funcion2()
```

Parámetros con valores por defecto

Vamos a ver un ejemplo de funciones en Swift con parámetros con valores por defecto.

Como podemos ver, hemos creado una función llamada 'unirCadenas' a la que le pasamos dos parámetros: 'separador' del tipo Character y 'cadenas' del tipo Array de String.

Esta función nos devolverá un String. Aquí podemos ver que el parámetro 'separador' está igualado a un espacio en blanco. Esta es la forma correcta de hacer que un parámetro de una función tenga un valor por defecto. A partir de ahora, en el caso de que llamemos a la función 'unirCadenas' y no le pasemos el parámetro 'separador', su valor será automáticamente espacio en blanco.

Esta función lo que hace es que va a unir todos los String del Array 'cadenas' separándolos mediante el elemento 'separador', creando una nueva cadena y devolviéndola.

Aquí podemos ver que hemos llamado a nuestra función dos veces: una pasándole un separador que es el guión y otra sin pasarle ningún separador. Después mostramos el resultado de ambas llamadas.

Vamos a compilar nuestro ejemplo para ver si no tiene ningún error y lo ejecutaremos.

Se ha pasado la compilación correctamente y ahora lo ejecutamos.

```
El resultado 1 vale: valor1-valor2-valor3-
El resultado 2 vale: valor1 valor2 valor3
```

Aquí podemos ver que tenemos dos resultados:

los elementos valor1, valor2 y valor3 que le hemos pasado a la función, pero en el primer caso estarán separados mediante un guión y en el segundo mediante un espacio en blanco, ya que no hemos pasado el parámetro 'separador'.

Código

```swift
import Foundation

/
 * Esta función une varias cadenas con un separador indicado
 * @param separador Este es el separador para separar las cadenas. Si no
se indica se usará el espacio en blanco.
 * @param cadenas Array con las cadenas a unir
 * @return Devuelve una nueva cadena con el resultado de unir las cadenas
indicadas con el separador indicado
*/
func unirCadenas(separador:Character = " ", cadenas:[String]) -> String
{
    var union:String = ""
    for cadena in cadenas
    {
        union += cadena
        union += String(separador)
    }
    return union
}

let resultado1 = unirCadenas(separador: "-", cadenas: ["valor1",
"valor2", "valor3"])
let resultado2 = unirCadenas(cadenas:["valor1", "valor2", "valor3"])

// Mostramos los resultados
print("El resultado 1 vale: \(resultado1)")
print("El resultado 2 vale: \(resultado2)")
```

Clases
Structs e inicializadores

El lenguaje de programación Swift soporta, entre muchos otros, el paradigma de la programación orientada a objetos, con lo que podremos definir nuestras propias clases con su determinada funcionalidad para resolver los problemas de una forma mucho más sencilla.

Para definir una clase, utilizaremos la palabra reservada class y escribiremos todo su código correspondiente dentro de un **bloque**.

Lo primero que tendremos que hacer es definir las variables de la clase, que **no tendrán ningún modificador de visibilidad**, ya que será el propio Swift el que se encargue de toda la seguridad de este tipo. Deberemos indicarle su tipo cuando **declaremos** las variables.

Un elemento nuevo que nos ofrece Swift son las estructuras (structs), con las que podremos **agrupar una serie de variables**. Las estructuras y las clases son muy parecidas en Swift, pero con las **estructuras** podremos **definir** las **variables sin** necesidad de darles un **valor**.

Ejemplo de estructura y clase básica

```
struct calificaciones {
    var nota1, nota2, nota3: Double
    }
class Alumno {
    let nombre:String   // Nombre del alumno
    let edad: Int   //Edad del alumno
    let notas: calificaciones   // Calificaciones del alumno
}
```

Como ocurría en Java, deberemos crear los **constructores** de la clase para poder crear objetos de ella. En Swift, los constructores se llaman inicializadores, y solo podremos tener **uno** con los parámetros que necesitemos.

Normalmente, se suele construir **un inicializador con todas las variable**s de la clase.

Los inicializadores se crearán mediante la palabra reservada init y para acceder a las **variables de la propia clase**, deberemos usar self.

Ejemplo de inicializador en una clase

```
struct calificaciones {
    var nota1, nota2, nota3: Double
    }
class Alumno {
    let nombre:String   // Nombre del alumno
    let edad: Int   //Edad del alumno
    let notas: calificaciones   // Calificaciones del alumno

    // Inicializador de la clase
    init(nombre: String, edad: Int, notas: calificaciones) {
        self.nombre = nombre
        self.edad = edad
        self.notas = notas
    }
}
```

Inicializadores de conveniencia

Si recordamos cuando estudiamos las clases en Java, podíamos declarar tantos constructores como necesitásemos, concretamente, tantos como combinaciones diferentes pudiéramos hacer con las variables de la clase.

Hemos visto en el punto anterior que únicamente podremos crear un inicializador en nuestras clases, pero ¿y si necesitamos **tener que declarar varios inicializadores** para poder cumplir con ciertas funcionalidades de un proyecto?

Esto también es posible realizarlo en Swift mediante lo que se conoce como inicializadores de conveniencia. Con este tipo de inicializadores, podremos crear tantos tipos de inicializadores como necesitemos y que nos permitan construir los objetos de nuestras clases de una forma mucho más cómoda.

El concepto de inicializador de conveniencia es muy simple, lo que estaremos creando son **inicializadores alternativos** que van a **terminar invocando al inicializador principal**, pero dando la posibilidad de **preprocesar los datos** antes de la **llamada al** inicializador **principal**.

Para crear un inicializador de conveniencia, deberemos utilizar las palabras convenience init seguidas **entre paréntesis de los parámetros** que vayamos a pasarle. Una vez que hayamos procesado los datos de la forma adecuada, tendremos que **llamar al inicializador principal** de la clase mediante self.init.

Ejemplo de inicializador de conveniencia

```
struct calificaciones {
    var nota1, nota2, nota3: Double
    }
class Alumno {
    let nombre:String   // Nombre del alumno
    let edad: Int   //Edad del alumno
    let notas: calificaciones   // Calificaciones del alumno

    // Inicializador de la clase
    init(nombre: String, edad: Int, notas: calificaciones) {
        self.nombre = nombre
        self.edad = edad
        self.notas = notas
    }

    // Inicializador de conveniencia
    convenience init(nombre: String, edad: Int, notas1: Double, nota2:
Double, nota3: Double) {
        let notas = calificaciones(nota1: nota1, nota2: nota2, nota3:
nota3)
        self.init(nombre: nombre, edad: edad, notas: notas)

    }
}
```

Recordatorio sobre inicializadores

Recuerda que siempre que crees una clase en Swift, el **inicializador principal** de la misma deberá llevar como **parámetros todos los atributos**, para luego así poder tener libertad total de crear todos los **inicializadores de conveniencia que necesites**.

Heredando en Swift

El lenguaje de programación Swift es un lenguaje que **soporta la orientación a objetos**, y, como tal, va a poder soportar la herencia, pudiendo utilizarla como estamos acostumbrados de otros lenguajes (Java).

La herencia soportada por Swift es la herencia simple, que si recordamos, esto quería decir que una clase solo podrá heredar de otra, siendo una práctica mucho más segura que la herencia múltiple.

No obstante, Swift sí que nos proporciona una funcionalidad extra que puede sustituir a la herencia en algunas ocasiones. Esta es la extensión de clases.

La extensión de clases nos aporta la posibilidad de poder agregarle una funcionalidad extra a una clase ya creada, es decir, podemos hacer que esa clase “se extienda” y nos ofrezca más funcionalidades. Esto también se podría resolver **creando una clase que heredara de otra** e implementándole la **nueva funcionalidad** a la nueva clase. Con esto, tenemos que tener en cuenta que el uso de la herencia en Swift es **mucho más específico** que en lenguajes como Java.

Tuplas

Las tuplas son un concepto que nos proporciona el lenguaje de programación Swift, aunque también es cierto que las podemos encontrar en otros lenguajes.

Estas son una **agrupación de valores** que **no tienen por qué ser del mismo tipo** de datos, sino que podremos mezclar enteros con cadenas, con reales, etc.

Para poder declarar una variable del tipo tupla, tendremos que indicar todos sus elementos entre paréntesis y separados mediante comas:

```
var tupla = ("palabra", 100, -9.6)
```

En la tupla anterior, podemos ver que tenemos tres elementos dentro de ella, uno del tipo cadena, uno del tipo entero y el último del tipo real.

Las tuplas podremos mostrarlas mediante la instrucción print, presentando todos los valores de la misma al llevarlo a cabo.

```
var tupla = ("palabra", 100, -9.6)
print(tupla)
```

Resultado:

```
("palabra", 100, -9.5999999999999996)
```

También podremos acceder a los elementos de una tupla mediante su índice, por ejemplo, si accedemos a tupla.2 del ejemplo anterior, estaremos accediendo al valor -9.6.

Accediendo al valor de una tupla por su índice

```
var tupla = ("palabra", 100, -9.6)
print(tupla.2)
```

Resultado:

```
-9.6
```

No solo vamos a poder crear tuplas de tres elementos, sino que podremos crearlas con los **elementos que necesitemos**, pudiendo acceder a sus valores para **consultarlos o modificarlos** mediante sus índices.

Otra forma de declaración de tuplas que podremos utilizar será dándoles un nombre a cada uno de sus **componentes**, de esta forma, cuando queramos **acceder** a un valor de un elemento de la misma, lo podremos hacer **mediante su nombre**, no mediante su índice.

Accediendo a un valor de una tupla por su nombre de parámetro

```
var tupla: (dato1: String, dato2: Int) = ("hola", 2020)
print(tupla.dato2)
```

Resultado:

```
2020
```

Por último, cabe destacar que podremos devolver una tupla dentro de una función, haciendo que dicha función pueda devolver todos los valores que necesitemos en forma de tupla.

Función que devuelve una tupla de dos valores

```
func funcion() -> (valor1: String, valor2: Int) {
    return("Devuelvo", 4)
}
```

Enumerados en Swift

Exploraremos la definición y aplicación de enumerados en Swift. Aprenderemos cómo definirlos y utilizarlos de manera efectiva en nuestro código.

Contenido:

Definición de Enumerados:

Para definir un enumerado en Swift, utilizaremos la palabra reservada 'enum', seguida del **nombre del enumerado**. Dentro del bloque de código del enumerado, emplearemos la palabra 'case' para definir los **valores permitidos**.

Ejemplo Práctico:

Consideremos un enumerado llamado 'Movimiento', cuyos valores pueden ser: derecha, izquierda, arriba y abajo. Podremos declarar **variables de tipo 'Movimiento'** de la siguiente manera:

```
var movi = Movimiento.DERECHA // Declaración de una variable
```

Si deseamos asignarle el valor "ARRIBA", la sintaxis sería:

```
movi = Movimiento.ARRIBA
```

Visualización del Movimiento:

Utilizando la orden 'print', podemos mostrar el movimiento por pantalla. Una vez compilado correctamente nuestro ejemplo, procedemos a ejecutarlo.

Conclusión:

Hemos demostrado cómo definir, utilizar y visualizar enumerados en Swift, lo que proporciona claridad y organización a nuestro código, especialmente en casos donde hay conjuntos discretos de valores relacionados.

```
import Foundation
// Este enumerado representa las posibles direcciones de un movimiento
enum Movimiento
{
    case DERECHA, IZQUIERDA, ARRIBA, ABAJO
}
var movi = Movimiento.DERECHA
movi = Movimiento.ARRIBA
print("El movmiento es: \(movi)")
```

Test sobre diccionarios en Swift

¿Cuál es el resultado de ejecutar este código?

var tripulacion = ["Capitan": "Luis", "Carlos": "Simon"]

tripulacion = [:]

print(tripulacion.count)

Grupo de opciones de respuestas

- 9
- 5
- 0
- 12

El resultado de ejecutar este código sería:

0

Esto se debe a que:

- primero **se inicializa un diccionario llamado tripulacion** con dos elementos.
- Luego, se **redefine** el diccionario como un **diccionario vacío** con **[:]**.
- Finalmente, se imprime la cantidad de elementos en el diccionario, que en este caso es 0.

Por lo tanto, la respuesta correcta es **opción: 0**.

Test sobre arrays en Swift

¿Cuál será el resultado al ejecutar el siguiente código?

let nombres = ["Luis", "Isabel", "David", "María"]

if let nombre = nombres[1]

{

print("Hola (nombre)")

}

Grupo de opciones de respuesta

- El código no compila.
- Se mostrará por pantalla el mensaje "Hola Isabel".
- Se mostrará por pantalla el mensaje "Hola Luís".
- Al ejecutar la aplicación se lanzará un error.

Solución:

El código no compila:

Parece que el error proviene de intentar utilizar la unión opcional (if let) en un **valor que no es opcional**. El código corregido sería el siguiente:

```swift
let nombres = ["Luis", "Isabel", "David", "María"]

if nombres.indices.contains(1) {
    let nombre = nombres[1]
    print("Hola \(nombre)")
}
```

En Swift, la propiedad indices de un arreglo proporciona un **rango de los índices válidos** para ese arreglo. Esto significa que puedes usar indices.contains(_:) para verificar si un índice específico está **dentro del rango** válido del arreglo.

En este código, nombres.indices devuelve un rango de índices válidos para el arreglo nombres. Luego, utilizamos contains(1) para verificar si el índice 1 está dentro de ese rango. Si es así, accedemos al elemento del arreglo en ese índice y lo imprimimos. Esto evita el error.

Test sobre interpolación de cadenas en Swift

¿Cuál es el resultado al ejecutar este código?

func decirHola(a nombre: String) -> String

{

return "Hola (nombre)!"

}

print("(decirHola(a: "Francis"))")

Grupo de opciones de respuesta

- Se mostrará por pantalla el mensaje “Hola (Francis)”.
- El código no compila
- Se mostrará por pantalla el mensaje “Hola Francis”.
- Al ejecutar la aplicación se lanzará un error.

Solución:

El código tal como está no compilará correctamente debido a un error de sintaxis. La forma correcta de imprimir el resultado de la función decirHola(a:) con el argumento "Francis" sería utilizando la **interpolación de cadenas**. Así que la opción correcta sería:

El código no compila.

Para corregirlo, deberías cambiar la última línea del código para que se imprima correctamente utilizando la interpolación de cadenas:

```swift
func decirHola(a nombre: String) -> String{
    return "Hola \(nombre)!"
}
print(decirHola(a: "Francis"))
```

dando como resultado:

Hola Francis!

Actividad sobre cálculo del número feliz

Se dice que un **número natural es feliz** si cumple que, si sumamos los cuadrados de sus dígitos y seguimos el proceso con los resultados obtenidos, finalmente obtenemos uno (1) como resultado.

Por ejemplo, el número 203 es un número feliz ya que

$$22+02+32=1312+32=1012+02=1$$

Se dice que un número es feliz de grado k si se ha podido demostrar que es feliz en un máximode k iteraciones. Se entiende que una iteración se produce cada vez que se elevan al cuadradolos dígitos del valor actual y se suman. En el ejemplo anterior, 203 es un número feliz de grado3 (además, es feliz de cualquier grado mayor o igual que 3).

- Realizar un programa que nos diga si un número es feliz.
- Realizar un programa que nos diga si un número es feliz y su grado.

Solución

```
func esNumeroFeliz(_ numero: Int) - > Bool {
    var num = numero
    var historial = Set<Int>()

    while num ! = 1 && !historial.contains(num) {
        historial.insert(num)
        var sumaCuadrados = 0
        while num > 0 {
            let digito = num % 10
            sumaCuadrados += digito * digito
            num / = 10
        }
        num = sumaCuadrados
    }

    return num == 1
}

func gradoNumeroFeliz(_ numero: Int) - > Int? {
    var num = numero
    var historial = Set<Int>()
    var grado = 0
```

```
    while num ! = 1 && !historial.contains(num) {
        historial.insert(num)
        var sumaCuadrados = 0
        while num > 0 {
            let digito = num % 10
            sumaCuadrados += digito * digito
            num / = 10
        }
        num = sumaCuadrados
        grado += 1
    }

    return num == 1 ? grado : nil
}

// Ejemplo de uso
let numero = 203
if esNumeroFeliz(numero) {
    print("El número \(numero) es feliz.")
    if let grado = gradoNumeroFeliz(numero) {
        print("Grado de felicidad: \(grado)")
    }
} else {
    print("El número \(numero) no es feliz.")
}
```

Nota sobre ejecutables Switch

Cuando empezamos a programar en algún lenguaje, normalmente, tendemos a querer identificar cuanto antes la forma de crear un ejecutable de nuestros programas.

Si recordamos NetBeans, este nos ofrecía un botón con el cual podríamos crear un fichero .jar que era el equivalente a los ejecutables para la máquina virtual de Java. Igual pasaba con Android Studio, que nos ofrecía la posibilidad de crear una APK de nuestro proyecto para poder instalarlo en un dispositivo móvil, pero ¿ocurre lo mismo con Swift?

Swift es un **lenguaje interpretado**, por lo que, con este tipo de lenguajes, **no podremos crear un ejecutable** de los programas, ya que el funcionamiento de los mismos no lo permite.