Tema 1: Introducción a Swift

Índice

Presentación

Presentación

- Swift es un lenguaje moderno publicado por Apple en 2014
- Inicialmente creado para desarrollar aplicaciones en ecosistemas macOS y iOS, hoy en día puede utilizarse para crear software en prácticamente cualquier hardware capaz de ejecutar código
    - Las pruebas de este tema introductorio podemos hacerlas en un Playground Online de Swift como http://online.swiftplayground.run
- Adopta patrones modernos y está optimizado para obtener el mayor rendimiento con la mayor comodidad para el programador

Presentación

- Algunas características:
  - Tipado implícito
  - No es necesario el punto y coma ;
  - Las variables se inicializan antes de su primer uso
  - Comprobación y gestión avanzada de errores
  - Gestión automática de memoria con Automatic Reference Counting

Variables y Constantes

Variables y Constantes

- Las variables se declaran con la palabra var

```
var miVar : Int = 0
```

- Para las constantes se utiliza let

```
let miConst : Int = 15
```

  - No es necesario inicializar una constante inmediatamente, pero sí que tendremos que darle valor una única vez

- Es recomendable declarar con var sólo los valores que vayan a cambiar y emplear let para el resto

Inferencia de Tipos

- Asigna de manera automática los tipos sin tener que indicarlos explícitamente
- Tradicionalmente, haríamos algo así:

```
var numero1: Int = 1
var numero2: Double = 2.0
var cadena: String = “Hola
```

- Gracias a la inferencia de tipos, podemos hacer lo siguiente:

```
var numero1 = 1
var numero2 = 2.0
var cadena = “Hola
```

- El compilador es capaz de asignar los tipos correctos cuando le asignamos un valor a la variable

Colecciones: Arrays, Diccionarios y Tuplas

Colecciones: Array

- Colección de elementos **del mismo tipo**: Array<Tipo>
- Array<Tipo> puede escribirse: [Tipo]
- Ejemplo:

```
var númerosPrimos : [Int] = [2,3,5,7,11,13,17,19,23,29]
```

- Puede ser mutables (var) o inmutables (let)
- Si lo inicializas en el momento de su declaración, la inferencia de tipos hace su trabajo y no hay que especificar su tipo

```
var numerosPrimos = [2,3,5,7,11,13,17,19,23,29]
```

Colecciones: Array

- Algunas utilidades:
  - array.count - número de elementos en el array
  - array.isEmpty - true o false según esté vacío o no
  - array.append(e) - añade el elemento e al final
    - Otra forma: array += [e]
  - array.insert() - inserción en una posición concreta
  - array.filter() - devuelve una copia del array formada por los elementos de éste que cumplan las condiciones deseadas

Colecciones: Tuplas

- Secuencia ordenada de elementos del mismo o distinto tipo entre paréntesis y separados por comas

```
var tupla = (“Hola",1,3.14)
```

- Swift conoce el tipo de cada elemento de la tupla gracias a la inferencia de tipos, pero también podemos especificarlo

```
var tuplaExplícita : (String, Int, Double) = (“Hola”, 1, 3.14)
```

- Podemos acceder a sus elementos mediante su índice

```
var texto = tupla.0
```

Colecciones: Tuplas

- Para que sea más fácil acceder a los elementos, podemos darles un nombre

```
var tuplaConNombres = (cadena: "Hola", entero: 1, decimal: 3.14)
var numeroEntero = tuplaConNombres.entero
```

- Otra forma de hacerlo:

```
var otraForma : (cadena : String, entero: Int, decimal: Double) = ("Hola", 1, 3.14)
var numeroDecimal = otraForma.decimal
```

Colecciones: Tuplas

- Las funciones pueden devolver tuplas

```
func obtenerNombreYApellido (dni: String) -> (String?,String?){
    var nombre, apellido : String?
    // ...
    return (nombre, apellido)
}
```

Colecciones: Diccionarios

- Colección no ordenada de clave-valor
  - Todas las claves son del mismo tipo
  - Todos los valores son del mismo tipo
- Dictionary<TClave, TValor> o [TClave: TValor]
- Ejemplos:

```
var diccionario : Dictionary<String,String>
var otroDiccionario : [String : String]
var diccionarioConElementos = ["pi" : 3.14, "e" : 2.72]
```

Colecciones: Diccionarios

- Algunas utilidades:
  - diccionario.count - número de pares clave-valor en el diccionario
  - diccionario.isEmpty - true o false según esté vacío o no

Colecciones: Diccionarios

- Podemos añadir elementos

```
diccionarioConElementos["euler"] = 0.57
```

- Y modificar otros ya existentes

```
diccionarioConElementos["pi"] = 3.14159265359
```

Colecciones: Diccionarios

- Podemos acceder a los valores almacenados

```
diccionarioConElementos[“e"]
```

- Podemos eliminar elementos del diccionario:

```
diccionarioConElementos["e"] = nil
```

- O podemos obtener un valor y eliminarlo de un paso:

```
var pi = diccionarioConElementos.removeValueForKey("pi")
```

Colecciones: Diccionarios

- Podemos acceder a los valores almacenados

```
diccionarioConElementos[“e"]
```

- ¡Cuidado!: esto devuelve TValor?

Colecciones: Diccionarios

- También podemos iterar por los diccionarios

```
var letrasEÍndices = ["a": 1,"b": 2, "c": 3, "d": 4, "e": 5]

for (letra, posicion) in letrasEÍndices{
    print("Letra: " + letra + ", posición en el alfabeto: \(posicion)")
}
```

Opcionales

Opcionales

- Los valores de los datos en Swift son no opcionales por defecto
- Esto significa que debemos de asignar un valor no nil

```
class MiClase {
    var saludo: String = "Hola" // OK
    var nombre: String = nil // error de compilación
}
```

- Podemos solucionar el error producido por **nombre** de dos formas: asignando un valor no nulo o declarando **nombre** como opcional

Opcionales

- Para indicar que un valor es opcional, hay que poner una interrogación (?) después del tipo

```
var nombre: String? = nil // 👍
```

Opcionales

- Para indicar que un valor es opcional, hay que poner una interrogación (?) después del tipo

```
var nombre: String? // 👍

var 💩: String = “Caca 😀"
```

Opcionales

- Los opcionales previenen de errores en tiempo de ejecución añadiendo una comprobación extra en tiempo de compilación

```
var saludoCompleto = saludo + " " + nombre // error de compilación
```

- Nombre es del tipo String?, lo que quiere decir que puede contener un String o nil

- El error que nos da es: error: value of optional type 'String?' not unwrapped; did you mean to use '!' or '?'?

Opcionales: Unwrapping

- ¿Qué es el unwrapping?
  - String? ≠ String
  - Para acceder al valor que estamos envolviendo con el opcional, debemos hacer unwrapping
  - Para hacerlo, sólo hay que poner una exclamación (*forced unwrapping*) detrás del nombre del opcional
  - ¡Cuidado! En este punto podemos tener errores de ejecución
    - Esta funcionalidad replica la gestión de null de otros lenguajes como Java

```
var saludoCompleto = saludo + " " + nombre! // error de ejecución
```

Opcionales: Unwrapping

- Conviene asegurarse de que el opcional no sea nil antes de intentar acceder a su contenido

```
if (nombre != nil) {
    var saludoCompleto = saludo + " " + nombre! // 👌
}
```

Opcionales: Implicit Unwrapping

- Con el implicit unwrapping conseguimos hacer forced unwrapping automáticamente sin necesidad de utilizar el signo de admiración cada vez que queramos acceder al valor del opcional

- Para indicar que un valor tiene implicit unwrapping, utilizamos el signo de exclamación (!) en su declaración en lugar de la interrogación

```
var nombreImplicito: String! // 👏
```

Opcionales: Optional Binding

- Existe una forma más segura que el *forced unwrapping* llamada *optional binding*:

```
if let nombreSeguro = nombre {
    var saludoCompleto = saludo + " " + nombreSeguro // 👌
}
```

- Comprueba que nombre no sea nil
  - Si nombre es nil, no entra en el if
  - En caso contrario, asigna el valor desempaquetado de nombre a nombreSeguro y entra en el if

Opcionales: Optional Chaining

- Veamos la funcionalidad del *optional chaining* con un ejemplo un poco más complejo

```
class Persona {
    var nombre = "Juan"
    var coche : Coche?
}

class Coche {
    var matricula : String = "1234-ABC"
}

var juan = Persona()

var matriculaDeJuan = juan.coche?.matricula

juan.coche = Coche()

matriculaDeJuan = juan.coche?.matricula
```

Opcionales: Optional Chaining

- Veamos la funcionalidad del *optional chaining* con un ejemplo un poco más complejo

```
class Persona {
    var nombre = "Juan"
    var coche : Coche?
}

class Coche {
    var matricula : String = "1234-ABC"
}

var juan = Persona()

var matriculaDeJuan = juan.coche?.matricula // String?, en este caso es nil

juan.coche = Coche()

matriculaDeJuan = juan.coche?.matricula // ahora contiene un String
```

Opcionales: Nil-Coalescing

- El operador (a ?? b) devuelve el valor unwrapped de a si no es nulo o el valor de b si a es nulo, siendo a un valor opcional y b del mismo tipo que a!

```
c = a ?? b
```

- Es lo mismo que:

```
a != nil ? a! : b
```

Funciones

Funciones

- Sintaxis de las funciones:

```
func nombre (parametro1 : Int, parametro2 : String) -> Void {…}
```

- Las que no devuelven nada pueden escribirse así:

```
func nombre (parametro1 : Int, parametro2 : String) {…}
```

- Por defecto, los parámetros son constantes, pero podemos cambiarlo

```
func doblar (var num : Int) -> Int {
    num *= 2
    return num
}
```

Funciones

- De la forma por defecto, pasaríamos el parámetro por valor

```
var numero : Int = 3
doblar(numero)
print (numero) // Muestra 3
```

- Fuera de la función, numero queda inalterado
- Podemos forzar que se pasen por referencia

```
func doblarPorReferencia (num : inout Int) {
    num *= 2
}

var otroNumero = 3
doblarPorReferencia(num: &otroNumero)
print (otroNumero) // Muestra 6
```

Funciones

- En lenguajes tradicionales, puede no quedar del todo claro qué es cada parámetro ya que desde dentro de la función esos parámetros tienen un nombre por el que referirnos a ellos, pero no desde fuera

```
func division (a: Double, b: Double) -> Double {
        return a/b
}

division(4,2)
```

- En Swift, los parámetros tienen nombre, tanto desde fuera como desde dentro, llamados etiqueta del parámetro y nombre del parámetro (argument label y argument name). Cuando llamamos a una función, ponemos la etiqueta de cada parámetro delante de su valor.

```
func divide(numero: Double, entre:Double) -> Double {
    return numero/entre
}

divide(numero: 4, entre: 2)
```

- Por defecto, la etiqueta y el nombre son iguales, pero podemos cambiar este comportamiento:

```
func divide(numero a: Double, entre b:Double) -> Double {
    return a/b
}

divide(numero: 4, entre: 2)
```

- Si un parámetro tiene etiqueta, es obligatorio utilizarla al llamar a la función

Funciones

- Podemos darle a los parámetros valores por defecto

```
var num = 1

func incrementa(numero: inout Int, cantidad:Int = 1){
    numero += cantidad
}

incrementa(numero: &num)
print (num) // "2"

incrementa(&num, cantidad: 5)
print (numero: num) // “7"
```

- Los parámetros con valores por defecto deben de ir al final de la lista de parámetros

Funciones

- También pueden tomar un número indefinido de parámetros del mismo tipo

```
func sumatorio (sumandos:Int...) -> Int {
    var resul = 0
    for sumando in sumandos {
        resul += sumando
    }
    return resul
}

sumatorio() // 0
sumatorio(sumandos: 1) // 1
sumatorio(sumandos: 1,2,3) // 6
```

Closures

Closures

- Un closure es un bloque de código funcional autocontenido
- Similar a los blocks de Obj-C y a las lambda (Java, C#…)
  - Pueden capturar valores del contextos en que son definidos
- Las funciones son un tipo específico de closure
- Sólo vamos a ver un tipo de closure: *Closure Expression*

Closure Expressions

- Sintaxis:

```
{ (parámetros) -> TReturn in

    sentencias

}
```

- Ejemplo (mayor de dos números):

```
{(a: Int, b: Int) -> Bool in
    if a > b{
        return true
    } else {
        return false
    }
}
```

Closure Expressions

- Pueden pasarse como parámetros a una función

```
func maximo (numeros: [Int], mayor: (Int, Int) -> Bool) -> Int {
    var resul = numeros [0]
    for i in 1 ..< numeros.count{
        if mayor(numeros[i], resul){
            resul = numeros[i]
        }
    }

    return resul
}

var numeros = [-2,-1,0,1,2,3,4,5]

let mayor = maximo(numeros, {(a: Int, b: Int) -> Bool in
    if a > b{
        return true
    } else {
        return false
    }
})

println(mayor) // 5
```

Closure Expressions

- Este closure puede escribirse de forma más cómoda

```
let mayor = maximo(numeros, {(a: Int, b: Int) -> Bool in
    if a > b{
        return true
    } else {
        return false
    }
})
```

- Este closure puede escribirse de forma más cómoda

```
let mayorReducido = maximo(numeros, {(a,b) -> Bool in
    if a > b{
        return true
    } else {
        return false
    }
})
```

Closure Expressions

- ¡Y podemos reducirlo aún más!

```
let mayorMasReducido = maximo(numeros, {(a,b) -> Bool in a > b })
```

Closure Expressions

- ¡Y podemos reducirlo aún más!

```
let mayorMasReducido = maximo(numeros, {(a,b) -> Bool in a > b })
```

- Y un poco más…

```
let mayorTodaviaMasReducido = maximo(numeros, {$0 > $1})

println (mayorTodaviaMasReducido)
```

Closure Expressions

- Y más…

```
let mayorIrreducible = maximo(numeros, >)

println (mayorIrreducible)
```

Closure Expressions

- Podemos hacer un mínimo cambio para que la función nos devuelva ahora el elemento menor

```
let menorIrreducible = maximo(numeros, <)

println (menorIrreducible)
```