Módulo Swift [Swift básico]

Daniel Illescas Romero

KEEPCODING
Tech School

Índice

1. Introducción al lenguaje.
2. Tipos de datos básicos.
3. Declaración de variables y constantes:
   3.1. Mostrar y leer datos por consola.
4. Operadores.
5. Declaraciones condicionales.
6. Bucles.
7. Conversión y verificación de tipos.
8. Tipos en detalle:
   8.1. Colecciones, 8.2. Cadenas de texto,
   8.3. Tuplas, 8.4. Rangos,
   8.5. Valores opcionales/*nullables*, 8.6. Otros tipos: `Any`, `Data`, `URL`, `Date`.
9. Bibliografía y enlaces interesantes.

1. Introducción al lenguaje.

Swift es un lenguaje de programación compilado y estáticamente tipado, principalmente escrito en C++, creado por Apple y lanzado en 2014. Nace como el sucesor de Objective-C, lenguaje que apareció por primera vez en 1984. Swift ofrece bastantes características modernas (corrutinas, extensiones, valores opcionales...), rapidez, compatibilidad con múltiples sistemas operativos e interoperabilidad con Objective-C.

Uso:

- **Apps** en iOS, iPadOS, macOS, tvOS, watchOS.
- Servicios web con frameworks como **Vapor** (funciona en Linux y macOS).
- Soporte básico para Windows.

2. Tipos de datos básicos.

Todos los tipos de datos en Swift son objetos de un `struct` o `class`.

- Tipo de dato lógico (***Boolean***): `Bool` .
- Tipos numéricos:
  - **Naturales +** `0` (*unsigned integers* [*W Whole*]): `UInt` , `UInt8` , `UInt16` , `UInt32` , `UInt64` .
  - **Enteros** (*signed integers* [*Z Integers*]): `Int` , `Int8` , `Int16` , `Int32` , `Int64` .
  - **Números reales** (*R real*): `Float16` , `Float` (`Float32`), `Double` (`Float64`), `Decimal` .
- Rangos: `Range` , `ClosedRange` .
- Representación de texto: `Character` , `String` .
- Colecciones de valores: `Set` , `Array` , `Dictionary` .
- Tuplas (*tuples*): `(SomeType, SomeType, ...)` , ejemplo: `(Int, Int, String)` .
- Tipos opcionales: `Optional` .
- Otros tipos: cualquier valor: `Any` , array de bytes: `Data` , url: `URL` .

3. Declaración de variables y constantes.

En Swift es preferible declarar constantes en vez de variables siempre que no se quiera reasignar o cambiar valores. Las variables se definen con `var` y las constantes con `let`. Podemos omitir su tipo, ya que Swift lo infiere en base al valor, o indicarlo con `: Tipo`:

```
var age = 10
var name = "Daniel"
let heightInMeters: Double = 1.92
var heightOfBuilding: Double = 23_423 // o 23423. "_" es un separador opcional que podemos usar
let numbers = [1, 2, 3, 4, 5]
```

Para reasignar valores a las variables simplemente usamos el `=`:

```
age = 90
name = #"Other "quotes""# // Other "quotes". # se usa para raw strings
heightOfBuilding = 200.45
heightInMeters = 29.1 // ¡error de compilación! heightInMeters es una constante
print(age) // 90. print es la función en Swift para mostrar valores por consola.
```

3.1. Mostrar y leer datos por consola.

Para mostrar el valor de cualquier dato por consola usamos la función `print`, la cual acepta todos los valores que queramos como parámetros.

```
let name = "Daniel"
print(name) // output: Daniel
let age = 90
print(name, age) // output: Daniel 90
```

Para leer valores desde la consola usamos la función `readLine`, que devuelve una cadena de texto.

```
let userInput1: String? = readLine()
```

4. Operadores.

Según número de miembros sobre los que operan: unarios (1), binarios (2) o ternarios (3). Según la posición o lado al que se sitúen: prefijos, infijos y sufijos.

- **Operadores unarios**: `+`, `-`, `!`. Ejemplo: `+1`, `-10`, `valueOrNil!`, `!isEmpty`.
- **Operador de asignación**: `=`. Ejemplo: `variable = "value"`.
- **Operadores aritméticos**: `+`, `-`, `*`, `/`, `%`. Ejemplo: `10 + myAge`.
- **Operadores de comparación**: `==`, `!=`, `>`, `<`, `>=`, `<=`. Ejemplo: `a == b`.
- **Operadores lógicos**: NOT (`!`), AND (`&&`), OR (`||`). Ejemplo: `!isEmpty`, `isEmpty && isOrdered`.
- **Operadores de rango:** `...`, `..<`. Ejemplo: `0...10`, `6...`, `2..<10`.
- **Operadores de asignación compuestos**: `+=`, `-=`, `*=`, `/=`.
- ***Nil-Coalescing Operator***: `??`. Ejemplo: `a ?? b`, que es equivalente con: `a != nil ? a! : b`.
- ***Ternary conditional operator***: `(a ? b : c)`. Ejemplo: `(isEmpty ? 10 : 20)`.
- Otros: personalizados, operadores de bit a bit y aritméticos con *overflow*.

5. Declaraciones condicionales.

Disponemos de `if` para comprobar condiciones, `else if` para comprobar una condición en caso de no haber cumplido la principal y `else` cuando no se cumple ninguna. También existe el operador ternario `? :`.

```
let vegetaPowerLevel = 10_000
if vegetaPowerLevel > 9_000 {
    print("It's over 9000!")
} else if vegetaPowerLevel == 5 {
    print("Meh")
} else {
    print("other", vegetaPowerLevel)
}
print(vegetaPowerLevel > 9000 ? "over 9000!" :"meh") // ternary conditional operator
```

Además en Swift disponemos de `guard` para salir de forma temprana de una función en caso de no cumplir una condición.

```
guard name == "Daniel" else { return } // asegúrate que name es "Daniel", sino, sal
print(name, "is not Daniel")
```


5. Declaraciones condicionales.

Una forma sencilla de comprobar diferentes valores sin necesidad de añadir muchos `if/else if` es mediante `switch`. La diferencia con respecto a otros lenguajes es la gran potencia, flexibilidad y que no se entra en otros `case` si no ponemos `break`.

```
switch vegetaPowerLevel {
case 5:
        print("5")
case 70..<100: print("70-100", vegetaPowerLevel)
case 200...300: print("70-100", vegetaPowerLevel)
case 9001...:
        print("It's over 9000!")
case ...0:
        print("dead")
case _ where String(vegetaPowerLevel).count == 3:
        print("3 digit level")
default:
        print("other", vegetaPowerLevel)
}
```


6. Bucles.

Para bucles tenemos:

- `for-in`:

```
for i in 0..<10 { /* ... */ }
for number in numbers { /* ... */ }
```

- `while`:

```
while inputName.isEmpty { /* ... */ }
```

- `repeat-while`:

```
repeat {
    /* ... */
} while inputName.isEmpty
```


7. Conversión y verificación de tipos.

La forma estándar en Swift para conversiones entre tipos es mediante un constructor.

```
// Número a cadena
let number = 10
let string = String(number) // constructor del tipo String que acepta número

// Cadena a número
let string2 = "123"
let number2 = Int(string2) // nullable, puede fallar la conversión

// Conversión entre números
let integer: Int = 10
let double = Double(integer)
let integer2 = Int(double)
let heightInMeters = 1.75
let heightInMetersInteger = Int(heightInMeters) // = 1, ¡cuidado, valor truncado!
```


7. Conversión y verificación de tipos.

Para la verificación o casting de tipos disponemos de lo siguiente:

- `type(of:)` : devuelve el tipo de un valor.

```
print(type(of: 10)) // Int
let name = "A"
print(type(of: name))
```

- Palabra clave `is` : nos permite verificar si un valor es de un tipo.

```
let number: Any = 10
if number is Int { /* ... */ }
```

- Casting:

```
let integer = number as? Int // nullable, puede fallar el casting
```


8.1. Tipos en detalle: colecciones.

Cuando queremos almacenar varios valores, podemos usar las siguientes estructuras de datos:

- `Array<T>` o `[T]`: conjunto de valores (se pueden repetir) accesibles con los corchetes/subscript `[]`, permite añadir y borrar elementos dado un índice, buscar elementos, etc.
- `Set<T>`: conjunto de valores no repetidos y no ordenados, permite borrar un elemento específico dado su valor y encontrar un valor en el set de forma más rápida que un Array. Los elementos deben ser `Hashable`.
- `Dictionary<Key, Value>` / `[Key: Value]`: diccionario clave-valor, no ordenado, conocido en otros lenguajes como hashmap o map, permite representar una serie de propiedades con valores asociados. Rápido acceso y eliminación de valores dada cada clave única. Muy usado con `[String: Any]` cuando se deserializa un JSON. Las claves deben ser `Hashable`.

8.1. Tipos en detalle: colecciones.

```swift
var numbers: [Int] = [1,7,20,2] // también: let numbers: Array<Int> = [1,2,3]
numbers[0] // 1
numbers[2] // 20
numbers[0] = 10 // [10,7,20,2]
numbers.remove(at: 0) // [7,20,2]
numbers.append(9) // [7,20,2,9]
numbers.append(contentsOf: [4,5]) // [7,20,2,9,4,5]
numbers = [-1,-2] + numbers // [-1,-2,7,20,2,9,4,5]

var uniqueValues: Set<String> = ["Daniel", "Pepe", "Juan"]
uniqueValues.contains("Daniel") // true
uniqueValues.remove("Pepe") // ["Daniel", "Juan"] (sin orden concreto)
uniqueValues.insert("John") // ["Daniel", "Juan", "John"] (sin orden)

var levels: [String: Double] = ["Goku": 1e3, "Vegeta": 9000, "Yamcha": 0]
levels["Goku"] // 1000
levels["Frieza"] = 1e3 + 200 // 1200
levels.removeValue(forKey: "Yamcha")
print(levels) // ["Goku": 1000.0, "Vegeta": 9000.0, "Frieza": 1200.0]
```

8.2. Tipos en detalle: cadenas de texto.

Las cadenas de texto son conjuntos de caracteres. Literalmente:

```
let nameCharacters: [Character] = ["G", "o", "k", "u", " ", ":", ")"]
let name = String(nameCharacters) // "Goku :)"
```

Usando `String` podemos fácilmente borrar, insertar, añadir y buscar characters u otras cadenas de texto.

```
var text = " And this is to go... EVEN FURTHER BEYOND"
text.count // 40
text.append("!") // " And this is to go... EVEN FURTHER BEYOND!"
let thirdPosition = text.index(text.startIndex, offsetBy: 9) // 🤯😢😨. text[9] no!
text.insert(",", at: thirdPosition) // " And this, is to go... EVEN FURTHER BEYOND!"
text.contains("FURTHER") // true
text = text.trimmingCharacters(in: .whitespacesAndNewlines)
print(text) // "And this, is to go... EVEN FURTHER BEYOND!"
```

8.3. Tipos en detalle: tuplas.

Las tuplas son conjuntos finitos de valores que pueden ser del mismo tipo o de distinto. Permiten a las funciones devolver fácilmente varios valores sin crear un struct/class específico, permiten intercambiar fácilmente valores y las tuplas pueden tener parámetros con nombres.

```
let nameAndAge: (String, Int) = ("Vegeta", 46) // ("Vegeta", 46)
nameAndAge.0 // "Vegeta"
nameAndAge.1 // 46
let nameAndAge2 = (name: "Vegeta", age: 46) // (name "Vegeta", age 46)
nameAndAge2.name  // "Vegeta"
nameAndAge2.age  // 46

// tipo: (name: String, age: Int, height: Int)
let (aName, age, height) = (name: "Vegeta", age: 46, height: 164)
print(aName, age, height) // Vegeta 46

var a = 10
var b = 5
(b, a) = (a, b)
print(a, b) // 5 10
```

8.4. Tipos en detalle: rangos.

Los rangos son parte fundamental de Swift, se utizan para iterar en los bucles, para comprobar fácilmente si un valor está entre ciertos valores, obtener trozos de colecciones y generar valores aleatorios.

```
let range1 = 0..<10
let range2 = 5...10
let range3 = 20...
let range4 = ...40 // no iterable con for-in
let range5 = 1.45...2.20 // no iterable con for-in
let range6 = "a"..."z" // no iterable con for-in
print(range5.contains(1.7)) // true. También: range5 ~= 1.7
print(range6.contains("x")) // true

for value in range1 { print(value) } // 0 1 2 3 4 5 6 7 8 9
range2.forEach { print($0) } // 5 6 7 8 9 10

let numbers = [17,2,5,4,2]
numbers[2...4] // 5,4,2

(1...100).randomElement()
```

8.5. Tipos en detalle: valores opcionales/*nullables*.

Por defecto en Swift ningún valor puede ser nulo (`nil` en Swift), tan solo puede serlo si explícitamente marco el tipo como "Opcional", en código esto es marcar los tipos con una interrogación final `?` o como `Optional<Type>` . **Esto hace que solo tengamos que comprobar nulos cuando así lo requiera el tipo.**

```
var color: String = "#00FF00"
color = nil // ¡error de compilación!, color es un String no opcional/no nullable

var colorOrNil: String? = "red"
colorOrNil = nil // permitido
colorOrNil = "blue" // permitido

// Optional Chaining: necesario poner `?` al intentar llamar propiedades o métodos
// sobre objetos opcionales.
let colorNumberOfCharactersOrNil = colorOrNil?.count
print(colorNumberOfCharactersOrNil) // tipo: Int?
if let colorNumberOfCharactersOrNil {
    // si `colorNumberOfCharactersOrNil` no es nulo...
}
```

8.6. Otros tipos: `Any`, `Data`, `URL`, `Date`.

`Any` es un tipo especial en Swift que representa cualquier tipo de valor. Se usa principalmente al deserializar ficheros JSON que contienen valores de distintos tipos; permite también crear colecciones heterogéneas.

```
var anyValue: Any = 10
anyValue = "holaa!"
let things: [Any] = [2, 3, 6.54, "hi!"]
```

`URL` lo usamos para representar direcciones URL y `Data` para un listado de bytes. Súper útil en peticiones.

```
let url = URL(string: "https://jsonplaceholder.typicode.com/todos/1")!
let (data, response) = try await URLSession.shared.data(for:  URLRequest(url: url))
```

El tipo `Date` sirve para representar un instante de tiempo. `Calendar` nos ayuda calcular y comparar fechas.

```
var today = Date()
today.timeIntervalSince1970 // unix timestamp, segundos desde 1 enero 1970
```

9. Bibliografía y enlaces interesantes.

- Xcode: https://developer.apple.com/xcode/
- *The Swift Programming Language*: https://docs.swift.org/swift-book/documentation/the-swift-programming-language
- *Foundation framework*: https://developer.apple.com/documentation/foundation
- *Vapor*: https://github.com/vapor/vapor
- *Markup Formatting Reference*: https://developer.apple.com/library/archive/documentation/Xcode/Reference/xcode_markup_formatting_ref/
- https://kalkicode.com/

KEEPCODING

Tech School

Madrid | Barcelona | Bogotá

Datos de contacto

Daniel Illescas Romero
email: daniel.illescas.r@gmail.com