Swift

Elementos básicos

Sentencias, bloques y comentarios

Sentencias

- En Swift, las sentencias se escriben una en cada línea
- No es necesario incluir un `;` al final

Sentencias y ;

```
let cat = "🐱"; print(cat)
```

Bloques

- Agrupan instrucciones
- Definen el ámbito de las variables
- En Swift se utilizan las llaves `{` y `}` para delimitarlos

Comentarios

- De una línea, `//`
- De múltiples líneas, `/* */`
- Se pueden anidar, si están balanceados `/* /* */ */`

Variables y constantes

Declaración

```
let maximumNumberOfLoginAttempts = 10  // Constante

var currentLoginAttempt = 0  // Variable


var x = 0.0, y = 0.0, z = 0.0   // Múltiple
```

Tipos de datos básicos

Tipo	Descripción
`Int`	Valor numérico entero
`Float`	Valor numérico de precisión simple
`Double`	Valor numérico de precisión doble
`Bool`	Valor lógico, verdadero o falso
`Character`	Caracter individual
`String`	Cadena de texto

Inferencia de tipos

```
var welcomeMessage = "Hello"
```

Anotaciones de tipo

```
var welcomeMessage: String

welcomeMessage = "Hello"

var red, green, blue: Double
```

Salida por consola

```
print("This is a string")
```

```
var friendlyWelcome = "Hello!"

print("The current value of friendlyWelcome is \(friendlyWelcome)")
```

Valores numéricos y lógicos

Tipos de datos básicos

Tipo	Descripción
`Int`	Valor numérico entero
`Float`	Valor numérico de precisión simple
`Double`	Valor numérico de precisión doble
`Bool`	Valor lógico, verdadero o falso
`Character`	Caracter individual
`String`	Cadena de texto

Enteros

- Enteros con signo: `Int`
- Enteros sin signo: `UInt`
- Existen `Int8`, `Int16`, `Int32` e `Int64` (y las versiones sin signo)
- Al usar `Int`, Internamente la longitud cambia a `Int32` o `Int64` dependiendo de la plataforma (32 o 64 bits)

Límites de los enteros

```
let minValue = UInt8.min  // minValue is equal to 0, and is of type UInt8
let maxValue = UInt8.max  // maxValue is equal to 255, and is of type UInt8
```

Coma flotante

- Precisión simple: `Float` (32 bits, 6 dígitos decimales de precisión)
- Precisión doble: `Double` (64 bits, 15 dígitos decimales de precisión)

Literales numéricos

	Literal	Notación	Ejemplo
Enteros	Decimal	Sin prefijo	`1_000_000`
	Binario	`0b`	`0b1011`
	Octal	`0o`	`0o34`
	Hexadecimal	`0x`	`0xF3A`
Coma flotante	Decimal	`e`	`1.25e4`
	Hexadecimal	`p`	`0xAp3`

Conversiones de tipo

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine


let integerPi = Int(pi)  // Se trunca el valor
```

Valores lógicos

```
let orangesAreOrange = true

let turnipsAreDelicious = false
```

Cadenas de texto

Tipos de datos básicos

Tipo	Descripción
`Int`	Valor numérico entero
`Float`	Valor numérico de precisión simple
`Double`	Valor numérico de precisión doble
`Bool`	Valor lógico, verdadero o falso
`Character`	Caracter individual
`String`	Cadena de texto

String

```
let someString = "Some string literal value"
```

Cadena vacía

```
var emptyString = ""
var anotherEmptyString = String()


var noEstaVacia: String  // No está inicializada
```

Características de los String

- Si se declaran con `var` son mutables, si se declaran con `let`, no
- Se pueden concatenar con `+` y `+=`
- Se pueden recorrer los caracteres individuales accediendo a la propiedad `characters` con un `for-in`
- Son tipos por valor, se copian al pasarlos a funciones o asignarlos a otras variables
- Son compatibles al 100% con el `NSString` de Foundation

Interpolación de Strings

```
var friendlyWelcome = "Hello!"

print("The current value of friendlyWelcome is \(friendlyWelcome)")
```

Comparar Strings

- Se pueden comparar directamente con el operador `==`
- Disponen de `hasPrefix(_:)` y `hasSuffix(_:)` para comparar el principio o el final de la cadena

Utilidades para Strings

- Para contar los caracteres de una cadena accedemos a la propiedad `count` de `characters`
- Dispone de `append()` para añadir caracteres al final

Acceso a los caracteres individuales

```
let greeting = "Guten Tag!"

greeting[greeting.startIndex] // G
greeting[greeting.index(before: greeting.endIndex)] // !
greeting[greeting.index(after: greeting.startIndex)] // u

let index = greeting.index(greeting.startIndex, offsetBy: 7)
greeting[index] // a
```

Insertar caracteres

```
var welcome = "hello"
welcome.insert("!", at: welcome.endIndex)
// welcome now equals "hello!"

welcome.insert(contentsOf:" there".characters, at: welcome.index(before: welcome.endIndex))
// welcome now equals "hello there!
```

Eliminar caracteres

```
welcome.remove(at: welcome.index(before: welcome.endIndex))
// welcome now equals "hello there"

let range = welcome.index(welcome.endIndex, offsetBy: -6)..<welcome.endIndex
welcome.removeSubrange(range)
// welcome now equals "hello"
```

Tuplas

Tuplas

- Agrupan múltiples valores en uno
- Los valores pueden ser de cualquier tipo
- No tienen que ser del mismo tipo
- Permiten que una función devuelva varios valores agrupados
- Para agrupaciones complejas, hay que usar estructuras o clases, no tuplas

Tuplas

```
let http404Error = (404, "Not Found")

let (statusCode, statusMessage) = http404Error

print("The status code is \(statusCode)")
print("The status message is \(statusMessage)")
```

Tuplas

```
let (justTheStatusCode, _) = http404Error
print("The status code is \(http404Error.0)")
```

```
let http200Status = (statusCode: 200, description: "OK")
print("The status code is \(http200Status.statusCode)")
```

Variables opcionales

Optionals

- Permiten definir variables que pueden o no tener valor
- Se crean añadiendo **?** al tipo de dato de la variable

Optionals

```
let possibleNumber = "123"

let convertedNumber = Int(possibleNumber)
```

Optionals

```
var serverResponseCode: Int? = 404

serverResponseCode = nil  // Sin valor
```

Extraer el valor de un opcional

- Imprescindible: `Int` no es lo mismo que `Int?`
- Forced unwrapping: usando `!`
- Optional binding: para extraer el valor en un `if` o `while`
- Optional chaining: usando `?`, cuando trabajemos con propiedades de estructuras o clases

Forced unwrapping

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)

if convertedNumber != nil {
    print("\(possibleNumber) has an integer value of \(convertedNumber!)")
} else {
    print("\(possibleNumber) could not be converted to an integer")
}
```

Optional binding

```
let possibleNumber = "123"

if let actualNumber = Int(possibleNumber) {
    print("\(possibleNumber) has an integer value of \(actualNumber)")
} else {
    print("\(possibleNumber) could not be converted to an integer")
}
```

Opcionales implícitos

- Se declaran con `!` en vez de `?` en el tipo de dato
- No necesitan de `!` para acceder al valor, pero si no tienen valor disparar un error en tiempo de ejecución
- Se usan en la inicialización de clases con referencias `unowned`

Operador de coalescencia nil

- Se utiliza con opcionales mediante `??`
- Permite extraer el valor del opcional o si vale `nil`, un valor por defecto
- `a ?? b` es una abreviatura de `a != nil ? a! : b`

Operador de coalescencia nil

```
let defaultColorName = "red"
var userDefinedColorName: String?   // nil

var colorNameToUse = userDefinedColorName ?? defaultColorName
// colorNameToUse == "red"
```

Operador de coalescencia nil

```
userDefinedColorName = "green"

colorNameToUse = userDefinedColorName ?? defaultColorName
// colorNameToUse == "green"
```

Operadores: asignación y aritméticos

Operador de asignación

- Copia el contenido de la parte derecha en la parque izquierda
- Descompone los valores de las tuplas en variables individuales
- No devuelve valor
- Hay versiones compuestas, como +=

Operador de asignación

```
let b = 10
var a = 5
a = b
```

```
let (x, y) = (1, 2)
```

Operador de asignación

```
var y = 6
var x = y = 5  // Error

if x=5 {  // Error

}
```

Operadores aritméticos

Operador	Operación
`+`	Suma
`-`	Resta
`*`	Multiplicación
`/`	División
`%`	Resto de la división
`-i`	Menos unario (cambio de signo)
`+i`	Más unario (no afecta al valor)

Operadores aritméticos

- No soportan overflow o underflow, se produce un error de tiempo de ejecución
- Hay versiones con overflow, como &+
- La división y el resto entre 0 también provocan un error

Operadores aritméticos (con overflow)

Operador	Operación
&+	Suma
&−	Resta
&∗	Multiplicación