Swift 6

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 - No se puede modificar
var currentLoginAttempt = 0  // Variable - Se puede modificar


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

Anotaciones de tipo

```
var welcomeMessage: String

welcomeMessage = "Hello"

var red, green, blue: Double
```

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

Nombres

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

- No pueden contener espacios ni símbolos de flecha o bloques
- No pueden comenzar por número
- Admiten Unicode

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, 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)

Inferencia de tipos

```
let meaningOfLife = 42

let pi = 3.14159

let anotherPi = 3 + 0.14159
```

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
```

Crea una constante llamada `nombre` con tu nombre.

Crea una variable llamada `edad` y asígnale tu edad.

Incrementa la edad en 1 y muestra un mensaje con tu nueva edad usando `print`.

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

Cadenas de texto

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

Multiline String

```
let quotation = """
The White Rabbit put on his spectacles.  "Where shall I begin,
please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on
till you come to the end; then stop."
"""
```

Caracteres especiales

```
let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
// "Imagination is more important than knowledge" - Einstein

let dollarSign = "\u{24}"        // $,  Unicode scalar U+0024
let blackHeart = "\u{2665}"      // ♥,  Unicode scalar U+2665

let sparklingHeart = "\u{1F496}" // 💖, Unicode scalar U+1F496
```

Cadena vacía

```
var emptyString = ""               // empty string literal
var anotherEmptyString = String()  // initializer syntax

if emptyString.isEmpty {
    print("Nothing to see here")
}


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

Escribe un programa que:

- Tome un número entero y verifique si es par o impar.
- Imprima "Número par" o "Número impar".

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 con un **for-in**
- Son tipos por valor, se copian al pasarlos a funciones o asignarlos a otras variables

Recorrer los caracteres de un String

```
for character in "Dog!🐶" {
    print(character)
}
```

Concatenación de Strings

```
let string1 = "hello"
let string2 = " there"
var welcome = string1 + string2

var instruction = "look over"
instruction += string2

let exclamationMark: Character = "!"
welcome.append(exclamationMark)
```

Interpolación de Strings

```
let multiplier = 3
let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"
```

Recuento de caracteres

```
let unusualMenagerie = "Koala 🐨, Snail 🐌, Penguin 🐧, Dromedary 🐪"
print("unusualMenagerie has \(unusualMenagerie.count) characters")
```

Crea una variable llamada `nombre` con tu nombre completo (por ejemplo: `"Juan Pérez"`).

Usa los métodos de Swift para:

1. Obtener el número de caracteres de la cadena `nombre`.
2. Convertir la cadena `nombre` a mayúsculas.
3. Convertir la cadena `nombre` a minúsculas.
4. Calcula el número de vocales de tu nombre completo.

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
```

Acceso a los caracteres individuales

```
greeting[greeting.endIndex] // Error
greeting.index(after: greeting.endIndex) // Error
```

```
for index in greeting.indices {
    print("\(greeting[index]) ", terminator: "")
}
```

Insertar caracteres

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


welcome.insert(contentsOf: " there", 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"
```

Subcadenas

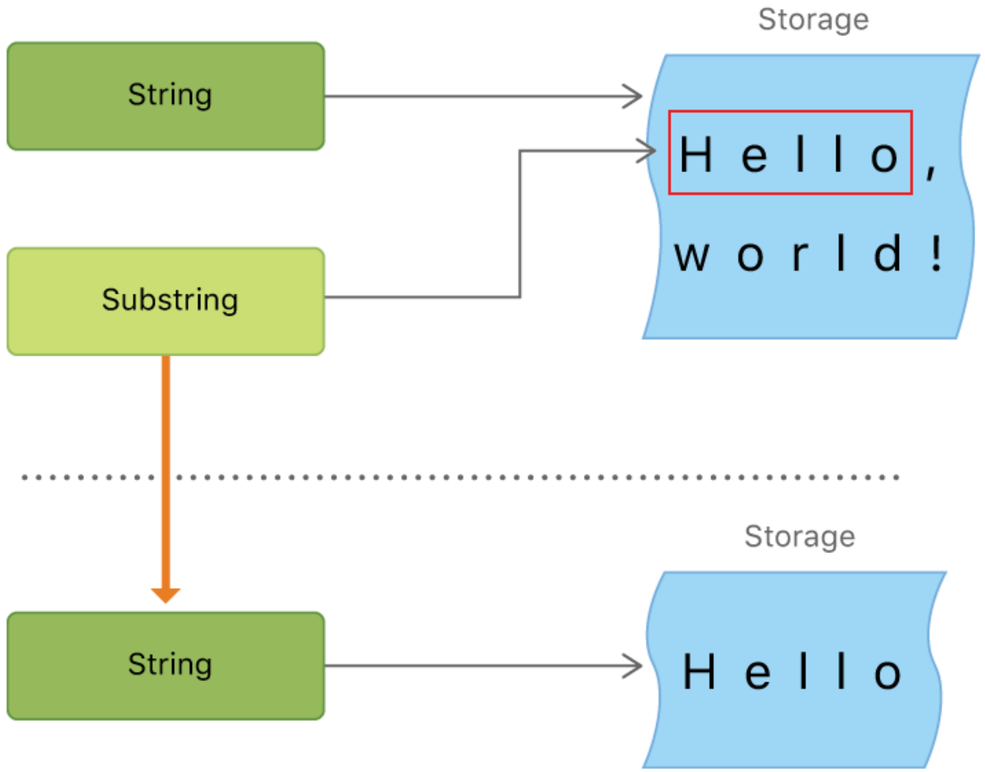
```
let greeting = "Hello, world!"
let index = greeting.index(of: ",") ?? greeting.endIndex //nil coalescing
let beginning = greeting[..<index]
// beginning is "Hello"

// Convert the result to a String for long-term storage.
let newString = String(beginning)
```

Subcadenas

```
let greeting = "Hello, world!"
let index = greeting.index(of: ",") ?? greeting.endIndex
let beginning = greeting[..<index]
// beginning is "Hello"

// Convert the result to a String for long-term storage.
let newString = String(beginning)
```



Comparar Strings

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

Igualdad de Strings

```
let quotation = "We're a lot alike, you and I."
let sameQuotation = "We're a lot alike, you and I."

if quotation == sameQuotation {
    print("These two strings are considered equal")
}
```

Crea una cadena con el nombre de tu ciudad (por ejemplo, `"Buenos Aires"`).

Extrae los primeros 6 caracteres usando la función `prefix()` y luego muestra la subcadena.

Usa `suffix()` para obtener los últimos 5 caracteres de la cadena.

Crea una cadena con el siguiente texto "Estoy aprendiendo Swift y desarrollando para iOS."

- Usa el método `contains()` para verificar si la cadena contiene la palabra `"Swift"`.

- Usa `range(of:)` para encontrar la posición de la palabra `"iOS"` en la cadena. Y después el método distance para convertir dicha posición en un índice numérico legible.

Crea una cadena que contenga una frase como `"El cielo es azul”`.

Usa el método replacingOccurrences(of:with:) para reemplazar la palabra "azul" por"rojo".

Tuplas

Tuplas

- Agrupan múltiples valores en uno
- Los valores pueden ser de cualquier tipo
- No tienen que ser del mismo tipo
- Las tuplas son útiles cuando quieres retornar múltiples valores desde una función.
- También cuando necesitas agrupar datos relacionados sin la necesidad de definir una estructura más compleja.

Tuplas

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

// Desempaquetado de tuplas
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 \(justTheStatusCode)")

print("The status code is \(http404Error.0)")
print("The status message is \(http404Error.1)")
```

Tuplas

```
let http200Status = (statusCode: 200, description: "OK")

print("The status code is \(http200Status.statusCode)")
print("The status message is \(http200Status.description)")
```

Tuplas

```
//Tupla como Clave de un Diccionario
let coordenadas = [
    (x: 1, y: 1): "Punto A",
    (x: 2, y: 3): "Punto B"
]

print(coordenadas[(1, 1)]) // "Punto A"
```

Crea una tupla `producto` que contenga el nombre del producto (String), su precio (Double) y su disponibilidad (Bool). Luego, imprime estos valores.

Variables opcionales

Optionals

- Permiten definir variables que pueden o no tener valor, de forma segura.
- Se crean añadiendo **?** al tipo de dato de la variable
- Un valor opcional puede contener un valor o `nil` (que indica "no hay valor")

Optionals

```
let possibleNumber = "123"

let convertedNumber = Int(possibleNumber) //Int?

//Solución posible
print(possibleNumber ?? 0)
```

Valor nil

```
var serverResponseCode: Int? = 404
serverResponseCode = nil

var surveyAnswer: String?   // nil
```

Extraer el valor de un opcional

- **OJO!**: **Int** no es lo mismo que **Int?**
- Opciones para desempaquetar opcionales:
  - Forced unwrapping: usando **!**
  - Optional binding y Guard let: para extraer el valor en un **if** o **while**
  - Optional chaining: usando **?**, cuando trabajemos con propiedades de estructuras o clases
  - Operador de coalescencia **??**

Forced unwrapping

```
//Forzar desempaquetado, OJO! Sólo si estás segur@ de que tiene valor

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

```
//Desempaquetado seguro: usa if let para comprobar si un opcional tiene un valor y desempaquetarlo en una variable temporal

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")
}
```

Optional chaining

/Acceder a propiedades/métodos/subíndices opcionales

```
class Persona {
    var direccion: Direccion?
}

class Direccion {
    var ciudad: String?
}

let persona = Persona()

// Usamos optional chaining para acceder a la ciudad y optional binding para
desempaquetar
if let ciudad = persona.direccion?.ciudad {
    print("La ciudad es \(ciudad)")
} else {
    print("No hay ciudad disponible.")
}
```

Guard let

//Usa `guard let` para desempaquetar opcionales en funciones o bucles donde el valor es necesario:

```
func imprimirNombre(_ nombre: String?) {
    guard let nombreSeguro = nombre else {
        print("No se proporcionó un nombre.")
        return
    }
    print("El nombre es \(nombreSeguro).")
}
imprimirNombre("Juan")
imprimirNombre(nil)
```

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

```
//Usa el operador de coalescencia nula (??) para proporcionar un valor
predeterminado si el opcional está vacío

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

var colorNameToUse = userDefinedColorName ?? defaultColorName
```

Operador de coalescencia nil

```
userDefinedColorName = "green"

colorNameToUse = userDefinedColorName ?? defaultColorName
```

Operador de coalescencia nil

Ejemplo de datos leídos por línea de comandos:

//Usando el operador coalescencia nil

```
var dato = Double(readLine()!) ?? 0.0
```

//Usando un optional binding

```
if let input = readLine(), let dato = Double(input) {
    print("El dato es: \(dato)")
} else {
    print("Entrada no válida. Se usará el valor por defecto: 0.0")
    let dato = 0.0
}
```

Crea una función que simule un sistema de inicio de sesión.

- La función recibirá dos opcionales: el nombre de usuario y la contraseña.

- Debe verificar si ambos valores son válidos y no nulos. Si falta alguno, debe retornar un mensaje indicando qué valor falta. Si ambos están presentes, valida si el nombre de usuario es `"admin"` y la contraseña es `"1234"`.

- Si las credenciales son correctas, retorna un mensaje de bienvenida, de lo contrario, indica que las credenciales son incorrectas.

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 i = 6
var j = i = 5  // Error

if j=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

```
let suma = 5 + 3        // 8
let resta = 5 - 3       // 2
let multiplicacion = 5 * 3 // 15
let division = 15 / 3   // 5
let modulo = 15 % 4     // 3
```

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

Errores comunes

- **División por Cero**: La división entre 0 genera un error en tiempo de ejecución

- **Tipos Numéricos Diferentes**: No puedes realizar operaciones aritméticas directamente entre tipos diferentes (e.g., `Int` y `Double`):

```
let entero: Int = 5
let decimal: Double = 2.5
let resultado = entero + decimal // ❌ Error de compilación.
```

Solución: Convierte explícitamente uno de los tipos:

```
let resultado = Double(entero) + decimal // ✅ Correcto
```

Crea una función que reciba dos números enteros y un operador como String (`+`, `-`, `*`, `/`, `%`) como parámetro, y retorne el resultado de la operación.