Introducción

Aplicaciones divertidas y avanzadas, redkaraoke, hrlog mostrador.

¿Qué es Swift?

Swift es un lenguaje de programación multiparadigma creado por [Apple](https://es.wikipedia.org/wiki/Apple) enfocado en el desarrollo de aplicaciones para iOS y macOS.

Swift playground

Introducción a Swift

Como manda la tradición ahí va nuestro hola mundo en Swift.

* print("Hola Mundo!")

Esta línea de código es un simple programa que se ejecuta sin necesidad de librerías en nuestro compilador y muestra por pantalla un hola mundo sencillo. El lenguaje es similar a muchos otros y no necesita puntos y comas para terminar las instrucciones.

Variables simples

Usamos let y var para hacer constantes y variables. El valor no hace falta darlo en tiempo de compilación. Las constantes son valores que se asignan al principio una vez y se usan en muchos sitios.

* var miVariable = 42
* miVariable = 50
* let miConstante = 42

Una constante o variable debe tener el mismo tipo que el valor al que se va a asignar, pero no siempre hay que asignarlo al principio. El tipo se puede obtener en tiempo de compilación. Por ejemplo myVariable es de tipo entero.

* let implicitoInteger = 70
* let implicitoDouble = 70.0
* let explicitoDouble: Double = 70
* let etiqueta = "The width is "
* let ancho = 94
* let anchoEtiqueta = etiqueta + String(ancho)

Podemos incluir variables en las cadenas de texto a imprimir

* let manzanas = 3
* let naranjas = 5
* let manzanasSummary = "I have \(manzanas) apples."
* let frutasSummary = "I have \(manzanas + naranjas) pieces of fruit."

Arrays y Diccionarios

Creamos Arrays y diccionarios usando los corchetes y accediendo a sus elementos por índice o claves.

* var listaCompra= ["carne", "agua", "fruta"]
* listaCompra [1] = "patatas"

Puedes añadir elementos a los arrays.

* listaCompra.append("blue paint")
* print(listaCompra)

Diccionarios:

* var ocupaciones = [
* "Carlos": "Capitan",
* "Pedro": "Mecanico",
* ]

ocupaciones ["Juan"] = "Enfermero"



O crearlos vacíos

* let emptyArray = [String]()
* let emptyDictionary = [String: Float]()

listaCompra = []

* ocupaciones = [:]

Contro de flujo

Ejemplo de bucle for para recorrer un array o colección

* let individualScores = [75, 43, 103, 87, 12]
* var teamScore = 0
* for score in individualScores {
* if score > 50 {
* teamScore += 3
* } else {
* teamScore += 1
* }
* }
* print(teamScore)
* // Prints "11"

Se puede usar if y let juntos para trabajar con valores que pueden haberse perdido. Esos valores se representan como opcionales. Una variable opcional puede contener un valor o nil que indica que se ha perdido el valor. Con una interrogación (?) después del tipo de una variable el valor es marcado como opcional.

* var optionalString: String? = "Hello"
* print(optionalString == nil)
* // Prints "false"
* var optionalName: String? = "John Appleseed"
* var greeting = "Hello!"
* if let name = optionalName {
* greeting = "Hello, \(name)"
* }
* Otra forma de dar valor por defecto a una variable opcional es el operador ??. Si se ha perdido el valor opcional el valor por defecto se usa en su lugar.
* let nickname: String? = nil
* let fullName: String = "John Appleseed"
* let informalGreeting = "Hi \(nickname ?? fullName)"

Comando Switch

* let vegetable = "red pepper"
* switch vegetable {
* case "celery":
* print("Add some raisins and make ants on a log.")
* case "cucumber", "watercress":
* print("That would make a good tea sandwich.")
* case let x where x.hasSuffix("pepper"):
* print("Is it a spicy \(x)?")
* default:
* print("Everything tastes good in soup.")
* }
* // Prints "Is it a spicy red pepper?"

Puedes usar for-in par iterar en un diccionario para usa cada pareja de clave-valor. Los diccionarios son colecciones sin ordenar, por lo que se devuelven sus clave sin ningún valor fijo.

* let interestingNumbers = [
* "Prime": [2, 3, 5, 7, 11, 13],
* "Fibonacci": [1, 1, 2, 3, 5, 8],
* "Square": [1, 4, 9, 16, 25],
* ]
* var largest = 0
* for (kind, numbers) in interestingNumbers {
* for number in numbers {
* if number > largest {
* largest = number
* }
* }
* }
* print(largest)
* // Prints "25"

Usar while para repetir un bloque de código en determinadas circunstancias.

* var n = 2
* while n < 100 {
* n \*= 2
* }
* print(n)
* // Prints "128"
* var m = 2
* repeat {
* m \*= 2
* } while m < 100
* print(m)
* // Prints "128"

También se pueden usar rangos para los bucles.

* var total = 0
* for i in 0..<4 {
* total += i
* }
* print(total)
* // Prints "6"

Funciones

Se usa func para declarar una función. Se llama a una función con la lista de parámetros entre paréntesis. Usa -> para separar el tipo de valor de retorno de la función.

* func saludar(persona: String, dia: String) -> String {
* return "Hola \(persona), hoy es \(dia)."
* }
* saludar (persona: "Javier", dia: "Miércoles")

Se puede usar una tuple para devolver más de un valor desde una función.

func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {

var min = scores[0]

* var max = scores[0]
* var sum = 0
* for score in scores {
* if score > max {
* max = score
* } else if score < min {
* min = score
* }
* sum += score
* }
* return (min, max, sum)
* }
* let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])
* print(statistics.sum)
* // Prints "120"
* print(statistics.2)
* // Prints "120"

Objetos y clases

Declaramos una clase en swift

**class** Forma {

**var** numeroDeCaras: Int = 0

**var** nombre: String

**init**(nombre: String) {

**self**.nombre = nombre

}

**func** simpleDescripcion() -> String {

**return** "Un \( nombre) es una forma con \( numeroDeCaras) caras."

}

}



Creamos una instancia de una clase

**var** forma = Shape(name: "Heptagon")

forma.numeroDeCaras = 7

**var** simpleDescripcion = forma. simpleDescripcion()

print(simpleDescripcion)



Utilizamos herencia y override para sobreescribir una función de la clase padre

* class Cuadrado: Forma {
* var longitudCara: Double
* init(longitudCara: Double, nombre: String) {
* self. longitudCara = longitudCara
* super.init(nombre: nombre)
* numeroDeCaras = 4
* }
* func area() -> Double {
* return longitudCara \* longitudCara
* }
* override func simpleDescripcion () -> String {
* return "Un cuadrado con caras de longitug \( longitudCara)."
* }
* }
* let test = Cudrado(longitudCara: 5.2, nombre: "my test square")
* test.area()
* test.simpleDescripcion()

Enumeraciones y estructuras

Crear enumeraciones

* enum Rank: Int {
* case ace = 1
* case two, three, four, five, six, seven, eight, nine, ten
* case jack, queen, king
* func simpleDescription() -> String {
* switch self {
* case .ace:
* return "ace"
* case .jack:
* return "jack"
* case .queen:
* return "queen"
* case .king:
* return "king"
* default:
* return String(self.rawValue)
* }
* }
* }
* let ace = Rank.ace
* let aceRawValue = ace.rawValue
* Usa struct para crear una estructura. Las estructuras soportan métodos y variables. Una de las mayores diferencias es que las estructuras se copian cuando son pasadas alrededor del código, mientras que las clases se pasan por referencia.
* struct Card {
* var rank: Rank
* var suit: Suit
* func simpleDescription() -> String {
* return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
* }
* }
* let threeOfSpades = Card(rank: .three, suit: .spades)
* let threeOfSpadesDescription = threeOfSpades.simpleDescription()

Manejo de errores

Representamos los errores con el protocolo Error.

* enum PrinterError: Error {
* case outOfPaper
* case noToner
* case onFire
* }

Usamos throw para lanzar un error. Si lanzas un error en una función, la función retorna inmediatamente y el código que maneja el error.

* func send(job: Int, toPrinter printerName: String) throws -> String {
* if printerName == "Never Has Toner" {
* throw PrinterError.noToner
* }
* return "Job sent"
* }

Una forma de manejar el error es do-catch. Dentro del bloque se marca el código que puede lanzar el error con try. Dentro del bloque catch se puede manejar el error.

do {

* let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")
* print(printerResponse)
* } catch {
* print(error)
* }
* // Prints "Job sent"
* Se pueden manejar errores específicos.

do {

* let printerResponse = try send(job: 1440, toPrinter: "Gutenberg")
* print(printerResponse)
* } catch PrinterError.onFire {
* print("I'll just put this over here, with the rest of the fire.")
* } catch let printerError as PrinterError {
* print("Printer error: \(printerError).")
* } catch {
* print(error)
* }
* // Prints "Job sent"