**SWIFT**

-No valores nulos.

Var name2:String? Con este símbolo indicamos que todavía no le vamos a asignar un valor.

Print(name?? “No hay nombre”) Asignamos valor por defecto.

- Diferenciar entre **variables y constantes.**

Ej. let name = “Ana”

Var age = 45

Podemos omitir el tipo de variable ya que Swift lo induce. Ej está que es de tipo int.

-**Tipo de estructura.**

Se utiliza camelcase para los nombres compuestos.

-**Operadores aritméticos.**

Importante no mezclar datos, ej int con un double.

**-Operadores lógicos.**

Var age= 20

If age <=35

{

Print (“Me encanta la música de Quevedo”)

}else{

Print(“Quevedo fue un gran escritor español”)

}

**-Interpolación de strings**

Crear un nuevo string a partir de otras variables o constantes

Var price= 600

Print (“El ordenador cuesta \(Price) euros”)

Let text2= “Hello”

Var text3 = “world”

Print(text2 +” “ + text3

Let text2= “Hello”

Var text3 = “”

Print(text2, text3)

If text3.isEmpty{

Print (“El texto esta vacio”)

}else{

Print (“El texto tiene contenido”)

}

-**Operadores de rango**

Existe figura de rango, idóneos para bucle so manipular ciertos elementos de una lista.

Let range = 1…10// rango que incluye tanto el 1 como el 10

For elt in range {

Print (elt) }

Imprime los números del 1 al 10.

Let openRange= 1..<10 //rango que incluye el 1 pero excluye el 10

For elt in openRange {

Print(elt)

} //Imprime numeros del 1 al 9.

**-Bucles**

Bucle for

Let vehicles = [“car”, “bus”, “bicycle”]

For I in 0..2 {

Print(vehicles[i])

}

Bucle while

Var newUsers= 0

While newUsers <100 {

newUsers +=1 //No existe incremento ++, hay que indcarlo como +=1

if newUsers ==50 {

print (“Mitadl del aforo completo”)

}

} print (“Aforo completo”)

**-Sentencias switch**

Var temp =14

Switch temp {

Case -5..10:

Print (“Hace frio”)

Case 11..20:

Print (“Hace bueno”)

Case 21..30: print (“Hace calor”)

Default: //Cuidado, es obligatorio poner el caso default

If temperatura <-5 {

Print (“Hace mucho frio”)

}else{

Print(“Hace mucho calor”)

}

}

**Operador ternario**

Cuando queremos asignar un valor a una variable/constante que depende de una condición. Estructura : condición ? caso positivo: caso negativo.

Ej. var Marketplace = “Spain”

url = Marketplace == “Spain” ? [www.amazon.es](http://www.amazon.es) : [www.amazon.it](http://www.amazon.it)

**Arrays. También infieren el tipo.**

//Definicion basica

var myArray = [1,2,3,4,5,6]

//Definición de array vacío:

var myArray2: [Int] = [] //array vacio

var myArray3 = [Int]() //array vacio

var myArray4:Int = [] //Error

//añadir elementos de uno en uno

myArray.append(7)

myArray.append(8)

//añadir varios elementos de golpe

myArray += [9,10,11,12]

//añadir elementos en una posición especifica

myArray.insert(0, at: 0)

print("Elementos de myArray: \(myArray)")

//contar elementos de un array

myArray.count

//comprobar si el array esta vacio

myArray.isEmpty

//eliminación de elementos de un array

myArray.removeAll() //Elimina todos los elementos

 myArray.removeFirst() //Elimina el primer elemento

myArray.removeLast() //Elimina el ultimo elemento

myArray.remove(at: 1) //Elimina el elemento en una posición especifica

//Recorrer un array

for elt in myArray{

    print(elt)

}

**Diccionarios**

Los diccionarios son colecciones de datos similares a los arrays con la salvedad de que los diccionarios se iteran por clave y no por índice:

var capitals = ["España": "Madrid", "Francia": "Paris", "Italia": "Roma"]

//forma 1 de recorrer un diccionario

for (k,v) in capitals{

    print("La capital de \(k) es \(v)")

}

//forma 2 de recorrer un diccionario

for elt in capitals{

    print("La capital de \(elt.key) es \(elt.value)")

}

//definición de un diccionario vacio

var dict = [String: Int]()

dict["clave1"] = 34

 dict["clave2"] = “cadena de texto” //Error

//eliminar elementos

dict.removeValue(forKey: "clave1") //borra un elemento de una clave

dict.removeAll() //borra todo

**Sets**

Los sets son parecidos a los arrays pero no admiten valores duplicados. Son útiles cuando queremos crear una lista de valores únicos:

var mySet = Set<Int>() //para indicar el tipo

mySet.insert(4)

mySet.insert(4) //no da error pero no lo duplica

print("El Set tiene ahora: \(mySet)")

//eliminar valores

mySet.remove(4) //indicamos el valor a eliminar, no la posición

**Funciones**

Las funciones son bloques de código que mantienen los programas organizados y modularizados.

               //funcion simple

func myFunc()

{

    print("Soy una funcion sencilla")

}

myFunc()

//funcion con parametros

func printAge(age: Int)

{

    print("Tengo \(age) años")

}

printAge(age:30)

//funcion con parametros que devuelve un valor

func sumNumbers(number1: Int, number2: Int)->Int

{

     return number1 + number2

}

let sum = sumNumbers(number1: 2, number2: 3)

print(sum)

//funcion con parametros anonimos que devuelve un valor

func sumNumbers2(\_ number1: Int, \_ number2: Int)-> Int

{

     return number1 + number2

}

let sum2 = sumNumbers2(2,3)

print(sum2)

//funcion con parametros con valores por defecto que devuelve un valor

func sumNumbers3(number1: Int, number2: Int = 10) -> Int

{

     return number1 + number2

}

let sum3 = sumNumbers3(number1:4)

print(sum3)

Aunque el uso de parámetros anónimos puede ser más cómodo, es recomendable definir el nombre de cada parámetro para que el código sea lo más explicativo posible de tal forma que cualquier persona que lea nuestro código entienda lo que hace sin necesidad de una documentación explicativa o comentarios.

**OPCIONALES**

Pueden tener un valor (habrá que desempaquetarlo) o ser nulos (dará error de ejecución)

Para desempaquetarla -> ‘if let’ o ‘guard let’ (la más común) O el uso de ? ? para proporcionarun valor predeterminado en caso de que el opcional sea nulo.

Ej if let

var name: String? = “Jorge”

if let name\_not\_nil = name{

    print(“El nombre es: \(name\_not\_nil)”)

} else{

    print(“El nombre es nulo.”)

}

Ejemplo con el operador “??”:

var name: String? = nil

let name\_not\_nil = name ?? “Jorge”

print(“El nombre es: \(name\_not\_nil)”)

**CLASES, SUBCLASES Y ESTRUCTURAS**

**Clases.**

Tipos de referencia en Swift. Cuando se asigna una instancia de una clase a otra variable, ambas variables apuntan a la misma instancia ne memoria. Se puede crear una clase que herede las propiedades y métodos de otra clase. Se utilizan cuando se necesita crear objetos complejos con comportamientos compartidos.

class Person{

    var name: String

    var age: Int

    init(name: String, age: Int) {

         self.name = name

         self.age = age

    }

    func greet() {

        print("Hola, mi nombre es \(name) y tengo \(age) años.")

    }

}

**Subclases**

Hereda propiedades y métodos de otra clase (superclase o clase padre) Se puede extender la funcionabilidad de una clase existente añadiendo nuevos métodos y propiedades o sobrescribiendo los existentes.

Estructura class (nombre subclase) : (nombre de la superclase dela que hereda).

class Student: Person {

    var school: String

    init(name: String, age: Int, school: String) { //Constructor de la clase Student.

         self. school = school

         super.init(name: name, age: age) //Llama al init de la clse padre person para inicializar los atributos heredados.

    }

    override func greet() { //Sobreescribe el método greet() de la clase Person usando override.

        print("Hola, soy \(name), tengo \(age) años y estudio en \(school).")

    }

}

**Estructuras**

Tipos de valor en Swfit, es decir que cuando se asigna un ainstancia de una estructura a otra variable, se copia el valor, en lugar de hacer referencia a la misma instancia de memoria. Las estructuras no admiten herencia. Tienen un inicializador autogenerado (construcotr) que permite inicializar las propiedades. Modelar datos simples o valores inmutables (coordenadas, fehcas, tamaños)

struct Point {

    var x: Int

    var y: Int

}

var point = Point(x: 10, y: 20)

point.x = 30

struct Person {

var name: String = “John Doe”

var age: Int

let dni: String

}

Var person1 = Person(name: “Jaime”, age: 45, dni: “000000Q”)

Print (person1)

**EXTENDER CLASE**

Util cuando se desea separar la funcionalidad adicional de la implementación principal.

Para extender una clase se utiliza ‘extension’ seguida del nombre de la clase que se desea extender

class Person{

    var name: String

    var age: Int

    init(name: String, age: Int) {

        self.name = name

        self.age = age

    }

    func greet() {

         print("Hola, mi nombre es \(name) y tengo \(age) años.")

    }

}

// Extensión de la clase Person

extension Person {

    func sayHobby(hobby: String) {

         print("Soy \(name) y uno de mis hobbies es  \(hobby).")

    }

}

// Uso de la clase y su extensión

let person = Person (name: "Juan", age: 30)

person.greet()

person.sayHobby(hobby: "deporte")

**TIPOS DE OPTIMIZACIÓN DE CÓDIGO**

EJ.

Anteponer constantes (let) sobre variables (var): Cuando se sabe que el valor no cambiará, es mejor usar constantes en lugar de variables.

var pi = 3.14 // Incorrecto

let pi = 3.14 // Correcto

Utilizar estructuras en lugar de clases para tipos de valor pequeños.

//Incorrecto

class Point {

    var x: Int

    var y: Int

    init(x: Double, y: Double){

       self.x = x

      self.y = y

   }

}

//Correcto

struct Point {

    var x: Int

    var y: Int

}

Utilizar programación funcional cuando sea apropiado.

// Incorrecto: Uso de bucle for para filtrar elementos

var evenNumbers: [Int] = []

for num in numbers {

    if num % 2 == 0 {

        evenNumbers.append(num)

    }

}

// Correcto: Uso de filter para filtrar elementos

let evenNumbers = numbers.filter { $0 % 2 == 0 }