

Tema 5: Programación funcional con Swift

Introducción

- Lenguaje principalmente imperativo, con conceptos funcionales (Haskell y Rust). Considerado **multi-paradigma**.
- **Fuertemente tipado** (necesario definir tipo de variables, funciones, etc.)

```
1 let n: Int = 10
2 let str: String = "Hola"
3 let array: [Int] = [1,2,3,4,5]
```

- **Inferencia de tipos** (El compilador permite identificar los tipos).

```
1 let n = 10
2 let str = "Hola"
3 let array = [1,2,3,4,5]
```

- Usa **var** para declarar **variables mutables**, y **let** para **variables inmutables**.

```
1 var x = 10
2 x = 20 // x es mutable
3 let y = 10
4 y = 20 // error: y es inmutable
```

*** Usaremos let (constantes) en paradigma funcional. Este paradigma nos permite evitar efectos laterales, lo que lo hace perfecto para entornos multi-hilo o multi-procesador. ***

Creación de nuevas estructuras y mutación

- En programación **funcional** debemos usar los métodos que **no modifican las estructuras**. Sino que crean nuevas estructuras y las devuelven.

```
1 // Código recomendable en programación funcional
2 let miArray = [10, -1, 3, 80]
3 let array2 = miArray + [100]
4 print(array2)
5 // Imprime:
6 // [10, -1, 3, 80, 100]
```

*** + crea un nuevo array resultado de concatenar miArray y el elemento 100 ***

- En programación **imperativa** usaremos los métodos que **modifican las estructuras** originales.

```
1 // Código no recomendable en programación funcional
2 var miArray = [10, -1, 3, 80]
3 miArray.append(100)
4 print(miArray)
5 // Imprime:
6 // [10, -1, 3, 80, 100]
```

*** append añade al array original miArray el elemento 100 ***

Funciones

Definición: `func nombre(nombreArg: tipo) -> tipoDevuelto`

```
1 func saluda(nombre: String) -> String {
2     let saludo = "Hola, " + nombre + "!"
3     return saludo
4 }
```

Para invocar a la función `saluda(nombre:)`:

```
1 print(saluda(nombre:"Ana"))
2 print(saluda(nombre:"Pedro"))
3 // Imprime "Hola, Ana!"
4 // Imprime "Hola, Pedro!"
```

Etiquetas de argumentos y nombres de los parámetros

```
func saluda(nombre: String, de ciudad: String) -> String {
    return "Hola \((nombre)! Me alegro de que hayas podido visitarnos desde \((ciudad))."
}
print(saluda(nombre: "Bill", de: "Cupertino"))
// Imprime "Hola Bill! Me alegro de que hayas podido visitarnos desde Cupertino."
```

La **etiqueta** del parámetro (el que usamos **al invocar la función**) es de y el **nombre** interno (el que se usa en el **cuerpo de la función**) es ciudad.

Si no se quiere una etiqueta del argumento para un parámetro, se puede escribir un **subrayado** (`_`) en lugar de una etiqueta.

```
func divide(_ x:Double, entre y: Double) -> Double {
    return x / y
}

print(divide(30, entre:4))
```

Parámetros y valores devueltos

- Podemos crear una función **sin parámetros**.
- Podemos **devolver múltiples valores** con una **tupla**. Accedemos a sus valores con su posición, o etiqueta.

```
func ecuacion(a: Double, b: Double, c: Double) -> (pos: Double, neg: Double) {
    let discriminante = b*b-4*a*c
    let raizPositiva = (-b + discriminante.squareRoot()) / 2*a
    let raizNegativa = (-b - discriminante.squareRoot()) / 2*a
    return (raizPositiva, raizNegativa)
}
```

```
let resultado = ecuacion(a: 1, b: -5, c: 6)
print("Las raíces de la ecuación son \"(resultado.0)\" y \"(resultado.1)\")
//Imprime "Las raíces de la ecuación son 3.0 y 2.0"
print("Las raíces de la ecuación son \"(resultado.pos)\" y \"(resultado.neg)\")
//Imprime "Las raíces de la ecuación son 3.0 y 2.0"
```

Recursión

Ej: Crea una función suma(hasta:) que devuelve la suma desde 0 hasta el número que le pasamos como parámetro.

```
func suma(hasta x: Int) -> Int {
  if x == 0 {
    return 0
  } else {
    return x + suma(hasta: x - 1)
  }
}
```

```
print(suma(hasta: 5))
// Imprime "15"
```

$$\begin{array}{c} \text{suma}(5) = 0 + 1 + 2 + 3 + 4 + 5 \\ \downarrow \qquad \qquad \downarrow \\ 5 + \text{suma}(4) \\ \downarrow \\ 5 + 4 \\ \downarrow \\ 5 + 3 + 2 + 1 + 0 \\ \downarrow \\ 5 + 4 + 3 + 2 + 1 + 0 = 15 \end{array}$$

Ej: Definir la función recursiva que suma los valores de un Array.

$$\begin{array}{c} \text{suma}([1, 4, 2, 5]) = 1 + 4 + 2 + 5 = 12 \\ \downarrow \qquad \qquad \downarrow \\ 1 + \text{suma}([4, 2, 5]) \\ \downarrow \qquad \qquad \downarrow \\ 1 + 4 + \text{suma}([2, 5]) \\ \downarrow \qquad \qquad \downarrow \\ 1 + 4 + 2 + \text{suma}([5]) \\ \downarrow \qquad \qquad \downarrow \\ 1 + 4 + 2 + 5 = 12 \end{array}$$

```
func sumaValores(_ valores: [Int]) -> Int {
  if (valores.isEmpty) {
    return 0
  } else {
    let primero = valores[0] //Guardamos el primer valor
    let resto = Array(valores.dropFirst()) //Guardamos el resto del
array
    return primero + sumaValores(resto)
  }
}
print(sumaValores([1,2,3,4,5,6,7,8])) //Imprime "36"
```

$$\begin{aligned} \text{MM}([1 \ 3 \ 4 \ 6]) &= (1, 6) \\ \downarrow \\ \text{MM}([3 \ 4 \ 6]) &= (3, 6) \\ \downarrow \\ (\text{M}(1, 3), \text{M}(1, 6)) \end{aligned}$$

```
let resultado = minMax(array: [1, 3, 4, 6])
print("El resultado es minimo \(resultado.min) y maximo \(resultado.max)")
//Imprimiría "El resultado es minimo 1 y maximo 6"
```

Tipos de función

- Objetos de primera clase

Funciones que reciben otras funciones

```
func sumaDosInts(a: Int, b: Int) -> Int {  
    return a + b  
}  
func printResultado(funcion: (Int, Int) -> Int, _ a: Int, _ b: Int) {  
    print("Resultado: \"(funcion(a, b))\"")  
    //no tiene en cuenta etiquetas a: b:  
}  
printResultado(funcion: sumaDosInts, 3, 5)  
// Prints "Resultado: 8"
```

Funciones en estructuras

```
var funciones = [identidad, doble, cuadrado]    //tipo [(Int) -> Int]  
print(funciones[0](10)) // 10  
print(funciones[1](10)) // 20  
print(funciones[2](10)) // 100
```

Funciones que devuelven otras funciones

```
func construyeSumador(inc: Int) -> (Int) -> Int {  
    func suma(x: Int) -> Int {return x+inc}  
    return suma    //Expresión devuelta clausura  
}  
  
var f2 = construyeSumador(inc: 10)  
var f3 = construyeSumador(inc: 100)  
print(f2(20))  
// Imprime "30"  
print(f3(20))  
// Imprime "120"
```

Tipos

Tipos con nombre

- Tipo al que podemos dar un nombre determinado cuando se define.
Ej: nombres de clases, de estructuras, de enumeraciones, de protocolos.

Tipos compuestos

- Son tipos sin nombre. En Swift se definen dos: tuplas y tipos función.
- Usamos **typealias** para dar un nombre a cualquier otro tipo.

Ej:

```
typealias Resultado = (Int, Int)

func quiniela(partido: Resultado) -> String {
    switch partido {
        case let (goles1, goles2) where goles1 < goles2:
            return "Dos"
        case let (goles1, goles2) where goles1 > goles2:
            return "Uno"
        default:
            return "Equis"
    }
}
```

Tipos valor y tipos referencia

- En **POO** tenemos dos construcciones: **estructuras y clases**.
- Las **estructuras** tienen una **semántica de copia** (son **tipos valor**) y las **clases** tienen una **semántica de referencia** (son **tipos referencia**).

1. Tipo valor

- ❑ Evitan efectos laterales.
- ❑ Facilitan gestión de la memoria. Elimina mem. al acabar un ámbito.

Ej:

```
var str1 = "Hola"
var str2 = str1
str1.append("Adios")
print(str1) // Imprime "HolaAdios"
print(str2) // Imprime "Hola"
```

2. Tipo referencia

- ❑ Múltiples variables guardan una referencia a la misma instancia.
- ❑ Efectos laterales

Enumeraciones

- Definen un tipo con un valor restringido de posibles valores
- Se **obtiene el valor** escribiendo el **nombre de la enumeración, un punto y el valor** definido. Si el tipo de enumeración se puede inferir no es necesario escribirlo.

Ej:

```
enum Direccion {  
    case norte  
    case sur  
    case este  
    case oeste  
}  
  
let direccionAIr = Direccion.sur  
switch direccionAIr {  
case .norte:  
    print("Nos vamos al norte")  
case .sur:  
    print("Cuidado con los pingüinos")  
case .este:  
    print("Donde nace el sol")  
case .oeste:  
    print("Donde el cielo es azul")  
}  
// Imprime "Cuidado con los pingüinos"
```

- Podemos **asignar** a las constantes del enumerado **un valor bruto** concreto de **un tipo** subyacente (Int, Char, String). **(El mismo valor para todas las instancias)**

Ej:

```
enum Quiniela: Int {  
    case uno=1, equis=0, dos=2  
}
```

- Puede **obtener el valor bruto** a partir del propio tipo o de una variable del tipo, usando **rawValue**.

Ej:

```
// Obtenemos el valor bruto a partir del tipo  
let valorEquis: Int = Quiniela.equis.rawValue  
  
// Obtenemos el valor bruto a partir de una variable  
let res = Quiniela.equis  
let valorEquis = res.rawValue
```

Enumeraciones instanciables

- Valores asociados a instancias de enumeraciones

Ej:

```
enum Prueba {  
    case num(Int)
```



```
}  
let valor1 = Prueba.num(10)    //valor 10 asociado a la instancia  
valor1  
let valor2 = Prueba.num(40)    //valor 40 asociado a la instancia  
valor2
```

- **Obtener el valor asociado** debemos usar una **expresión case let** en una sentencia **switch** con una variable a la que se asigna el valor.

Ej:

```
switch valor1 {  
case let .num(x):  
    print("Valor asociado al caso num: \(x)")  
}  
// Imprime "Valor asociado al caso num: 10"
```

Enumeraciones recursivas

- Preceder la palabra clave enum con **indirect**

Ej:

```
indirect enum Lista {  
    case vacia  
    case nodo(Int, Lista)  
}
```

Ej: Cómo crear instancias de enum. recursivas

```
let lista1 = Lista.nodo(30, Lista.vacia)  
let lista2 = Lista.nodo(20, lista1)  
let lista3 = Lista.nodo(10, lista2)  
0  
let lista: Lista = .nodo(10, .nodo(20, .nodo(30, .vacia)))
```

Ej: Diseña una función recursiva que sume los elementos de un enum lista con los casos: vacia, nodo (Int, Lista).

```
func suma(lista: Lista) -> Int {  
    switch lista {  
    case .vacia:  
        return 0  
    case let .nodo(car, cdr):  
        return car + suma(lista: cdr)  
    }  
}  
  
let z: Lista = .nodo(20, .nodo(10, .vacia))  
  
print(suma(lista: z))  
// Imprime 30
```

Opcionales

- Finalidad: promover la seguridad evitando excepciones causadas por el valor null.
- En Swift el valor nulo es **nil**
- No es posible asignar nil a una variable de un tipo normal.

- Para **usar nil** debemos **declarar** la variable usando un tipo opcional (**tipo?**)

```
var cadena: String? = "Hola"
cadena = nil
//String? Indica que podemos tener un valor nil o un valor del tipo
original.
```

- Para **obtener el valor** real de un tipo **opcional usamos !** (desenvoltura forzosa)

```
var x: Int? = 10
let y = x! + 10
print(y)
// Imprime "20"
```

Ligado opcional

- Es **obligado comprobar** si un **valor opcional es nil** si desconocemos el valor de la variable. **Usar un if.**

```
var x1: Int? = pedirNumUsuario()
var x2: Int? = pedirNumUsuario()

if let dato1 = x1, let dato2 = x2 {
    let suma = dato1+dato2
    print("Ningún nil y la suma de todos los datos es: \(suma)")
} else {
    print("Algún dato del usuario es nil")}
```

Operador nil-coalescing (??)

- Asigna un valor por defecto en una asignación si el opcional es nil

```
let a: Int? = nil
let b: Int? = 10
let x = a ?? -1
let y = b ?? -1
print("Resultado: \(x), \(y)")
// Imprime Resultado: -1, 10
```

Encadenamiento de opcionales

- Permite llamar a un método de una variable que contiene un opcional. Si es nil, devuelve nil, por el contrario se llama a la función.

```
let nombre1: String? = "Pedro"
```

```
let nombre2: String? = nil

let str1 = nombre1?.lowercased()
let str2 = nombre2?.lowercased()
// str1: String? = "pedro"
// str2: String? = nil
```

Definición de Lista con opcionales

```
indirect enum Lista {
    case nodo(Int, Lista?)
}
func suma(lista: Lista) -> Int {
    switch lista{
        case let .nodo(car, cdr):
            if (cdr != nil) {
                return car + suma(cdr)
            } else {
                return car
            }
    }
}

let lista1 . Lista = .nodo(1, .nodo(2, .nodo(3, .nodo(4, nil))))
print(suma(lista: lista1))    // Devuelve 10
```

Clausuras

Funciones definidas en el ámbito de otras funciones y devueltas como resultados.

Expresiones de clausuras

Permiten definir de forma compacta funciones que se pasan como parámetro de otras.

Ej: Usamos una clausura en la función `sorted(by:)`, que recibe una función (de dos parámetros) como parámetro para realizar la comparación entre elementos y ordenar una array. El array original no se modifica, se devuelve uno nuevo.

La clausura de ordenación devuelve `true` si el primer valor debería aparecer antes del segundo valor y `false` en otro caso.

- ❑ Sin usar expresión de clausura:

```
func primeroMayor(s1: String, s2: String) -> Bool {  
    return s1 > s2  
}  
  
let estudiantes = ["Kofi", "Abena", "Peter", "Kweku", "Akosua"]  
let alreves = estudiantes.sorted(by: primeroMayor)  
print(alreves)  
// Imprime ["Peter", "Kweku", "Kofi", "Akosua", "Abena"]
```

- ❑ Usando expresión de clausura:

```
let estudiantes = ["Kofi", "Abena", "Peter", "Kweku", "Akosua"]  
let alreves = estudiantes.sorted(by: { (s1: String, s2: String) ->  
    Bool in  
        return s1 > s2  
    })
```

Inferencia del tipo por el contexto

```
let alreves = estudiantes.sorted(by: { s1, s2 in return s1 > s2 } )
```

Devoluciones implícitas en clausuras con una única expresión

```
let alreves = estudiantes.sorted(by: { s1, s2 in s1 > s2 } )
```

Abreviaturas en los nombres de los argumentos

```
let alreves = estudiantes.sorted(by: { $0 > $1 } )
```

Funciones operadoras

```
let alreves = estudiantes.sorted(by: >)
```

Clausuras al final

- Pasar una expresión de **clausura** a una función **como el argumento final** de la clausura.

```
let alreves = estudiantes.sorted() { $0 > $1 }
```

```
let alreves = estudiantes.sorted { $0 > $1 }    //1 argumento, pueden omitirse los paréntesis
```

Valores capturados

Los ejemplos siguientes **no usan programación funcional**, porque la **variable capturada** por la clausura es una variable **mutable** (se ha definido con var y no con let).

```
func construyeIncrementador(incremento cantidad: Int) -> () -> Int {  
    var totalAcumulado = 0  
    func incrementador() -> Int {  
        totalAcumulado += cantidad  
        return totalAcumulado  
    }  
    return incrementador  
}
```

```
let incrementaDiez = construyeIncrementador(incremento: 10)  
incrementaDiez()  
// devuelve 10  
incrementaDiez()  
// devuelve 20  
incrementaDiez()  
// devuelve 30
```

```
let incrementaSiete = construyeIncrementador(incremento: 7)  
incrementaSiete()  
// devuelve 7
```

Clausuras con expresiones de clausura

```
func construyeIncrementador(incremento cantidad: Int) -> () -> Int {  
    var totalAcumulado = 0  
    return {totalAcumulado += cantidad  
            return totalAcumulado}  
}
```

Valores capturados y valores del ámbito local de ejecución

```
func construyeFunc() -> () -> Int {
  var x = 0
  return {
    x = x + 1
    return x
  }
}

let f = construyeFunc()      //f captura x = 0
print(f())                  // -> 1
print(f())                  // -> 2

func usaFunc(_ f: () -> Int) -> Int {
  var x = 10
  return f()
}

print(usaFunc(f)) // -> 3      //f sigue usando la referencia a x = 2

var x = 100
print(usaFunc {return x + 10}) // -> 110 //usaFunc usa la referencia a x = 100
```

Las clausuras son tipos de referencia

- Al asignar una función o una clausura a una constante o una variable, estamos realmente estableciendo que la constante o variable es una referencia a la función o la clausura.

```
let tambienIncrementaDiez = incrementaDiez
tambienIncrementaDiez()
// devuelve 50
```

FOS

Map -> func map<T>(_ transform: (Element) -> T) -> [T]

- Usada por estructuras como **Array**, Dictionary, Set o String.CharacterView

EJ: Implementar la función sumaParejas(parejas: [(Int, Int)]) -> [Int] que recibe el array de tuplas de dos enteros y devuelve un array con el resultado de sumar los dos elementos de cada pareja.

```
func suma(parejas: [(Int, Int)]) -> [Int] {  
    return parejas.map({(pareja: (Int, Int)) -> Int in  
        return pareja.0 + pareja.1})  
}  
suma(parejas: [(1, 1), (2, 2), (3, 3), (4, 4)])  
// devuelve [2, 4, 6, 8]
```

Filter -> func filter(_ isIncluded: (Element) -> Bool) -> [Element]

```
let numeros = [Int](0...10)  
numeros.filter {$0 % 2 == 0}  
// devuelve [0, 2, 4, 6, 8, 10]
```

Reduce -> func reduce<Result>(_ initialResult: Result,

_ nextPartialResult: (Result, Element) -> Result) -> Result

- Parecido al foldl de Scheme.
- Recibe como parámetro un valor inicial y una función de plegado que se aplica al resultado anterior y al elemento de la colección, devolviendo un resultado.

```
let cadenas = ["Patatas", "Arroz", "Huevos"]  
print(cadenas.reduce("*", {$0 + "-" + $1}))  
// Imprime "-Patatas-Arroz-Huevos"
```

Combinación de funciones de orden superior

```
let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
numeros.filter{$0 % 2 == 0}.map{$0*$0}  
// Devuelve el array [4, 16, 36, 64, 100]
```

```
let numeros = [103, 2, 330, 42, 532, 6, 125]  
numeros.filter{$0 >= 100}.reduce(0, +)  
// Devuelve 1090
```

Genéricos

- Generalizar las funciones para hacer que el código pueda **trabajar con cualquier tipo** usando función genérica.
- Puede aplicarse a funciones, enumeraciones, estructuras, clases, protocolos o extensiones.

Ej:

```
func intercambia(_ tupla: (Int, String)) -> (String, Int) {  
    let tuplaNueva = (tupla.1, tupla.0)  
    return tuplaNueva  
}
```

Usamos genéricos:

```
func intercambia<A,B>(_ tupla: (A,B)) -> (B, A) {  
    let tuplaNueva = (tupla.1, tupla.0)  
    return tuplaNueva  
}
```

Bibliografía

Los enunciados de los ejercicios resueltos, y los resúmenes, se han elaborado a partir del material publicado en <https://domingogallardo.github.io/> , material del que es propietario el Departamento de Ciencia de la Computación e Inteligencia Artificial de la Universidad de Alicante, Domingo Gallardo, Cristina Pomares, Antonio Botía y Francisco Martínez.