Swift

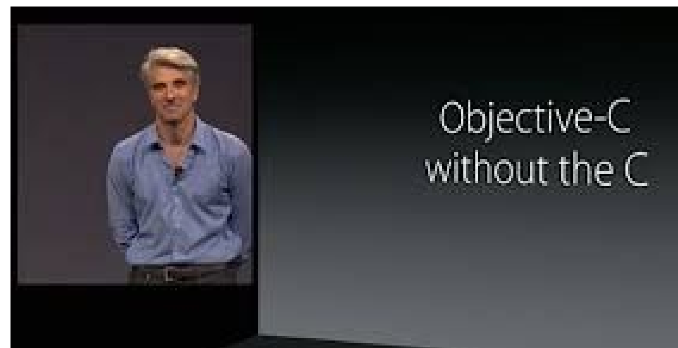
Angela María Muñoz Medina
Liseth Briceño Albarracín
Nicolás Larrañaga Cifuentes

¿Qué es Swift?

El nuevo lenguaje para iOS y OS X

- Es construido de lo mejor de C y objective C
- Presentado en el WWDC de 2014
- Soporta librerías de Objective-C y C
- Swift 3.0

https://swiftlang.ng.bluemix.net



TIPOS DE DATOS

- Int
  - Int32 o Int64
  - UInt32 o UInt64
- Float
- Double
- Bool
- String
- Character
- Optional

Optional: es un contenedor o referencia que puede almacenar nada o algo. Solo pueden usarse con un tipo variable colocando un '**?**'

```
1  var variable : String?
2
3  variable = "perro"
4
```

CONSTANTES Y VARIABLES

Constantes **let**

Variables **var**

```
1  var peso = 60
2  peso = 50
3  let altura = 43
```

```
2  var implicitInteger = 60
3  let implicitDouble = 60.5
4  let explicitDouble:Double = 60
```

INFERENCIA DE TIPOS

Swift es un lenguaje con **tipado estático**, es decir, las variables tienen un tipo establecido en tiempo de compilación. Pero si al declarar una variable le asignamos un valor inicial, el compilador puede inferir el tipo de la variable y no es necesario que lo indiquemos.

```
1  //Implicito
2  var mensaje = "Hello World!";
3
4  //Explicito
5  var mensaje : String = "Hello World!";
```

TIPOS POR VALOR O POR REFERENCIA

Tipos por valor

Cuando se realiza una asignación y el valor del tipo es copiado y manejado independiente del tipo original. Sucede lo mismo al pasar el tipo como función. Esto solo sucede con las estructuras, enumeración, y todos los tipos básicos de Swift: Int,Float,Booleans,etc.

Tipos por referencia

Cuando se realiza una asignación y el valor del tipo no es copiado, la nueva variable únicamente hace referencia al tipo original. Los cambios en el segundo se verán reflejados en el primero. Ej: Las clases

TIPOS POR VALOR O POR REFERENCIA

Tipos por valor

- El operador == realiza una comparación **por valor** (equivale al método equals de c# o equals de Java).

Tipos por referencia

- El operador === realiza una comparación **por referencia** (equivale al operador == de Java y generalmente al operador de == de C#)..

OPERADORES

Aritméticos:

- + -
- * / %
- +=, -=, *=, /=, %=

Logicos:

- NOT !a
- AND a && b
- OR a || b

De comparación:

- Igual ==
- Diferente !=
- Mayor que >
- Menor que <
- Mayor o igual >=
- Menor o igual <=
- Identidad (===) (!==) : si 2 referencias de objetos refieren a la misma instancia

OPERADORES

Especiales:

- Condicional ternario:

  “question ? answer1 : answer2”

- Coalescencia nula (nil):

  (a ?? b)

La cual nos permite evaluar una expresion opcional y en caso de que esta falle evaluar un valor por defecto.

OPERADORES

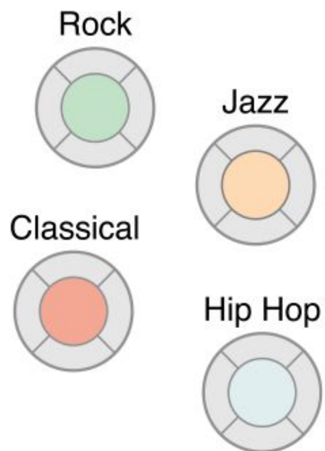
De rango:

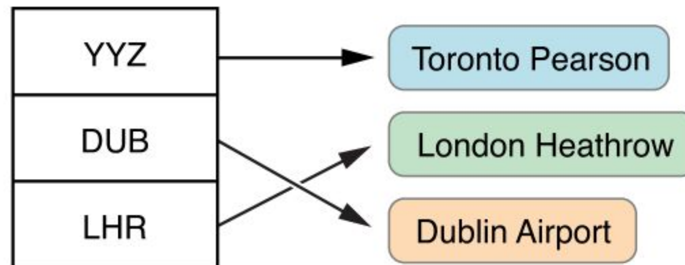
- Cerrado **a...b** : desde a hasta b e incluye valores de a y b
- Medio abierto **a..<b** : desde a hasta b pero no incluye b. Si a es igual a b el rango es vacío

COLECCIONES

Array

Indexes	Values
0	Six Eggs
1	Milk
2	Flour
3	Baking Powder
4	Bananas





Array

- Instanciamiento y acceso mediante []

Añadir

1. Se añaden elementos mediante **.append()**
2. Concatenando arreglos mediante el operador **+=**

```
1  var animales = ["vaca","perro", "cabra"]
2  var animales_vacio = [String]()
3  var animales_forzado : [String] = ["ballena","perro","gato"]
```

Array

```swift
//Crear arreglo vacio
var algunosAnimales = [String]()
//Imprimir cantidad de elementos
print("cantidad: \(algunosAnimales.count) animales.")
//Imprimir elementos
print(algunosAnimales)

//Agregar animales
algunosAnimales.append("hamster")
algunosAnimales += ["gato"]

//imprimir elementos
print(algunosAnimales)
```

```
cantidad: 0 animales.
[]
["hamster", "gato"]
iSwift:~ $
```

Array

---

3. Insertar un elemento en un índice específico mediante el método **insert(_,at:)**.

```
1  var animales = ["vaca", "perro", "cabra"]
2
3  animales.insert("Pez dorado", at: 0)
4  print(animales)
```

```
Swift Ver. 3.0 (Release)
Platform: Linux (x86_64)


["Pez dorado", "vaca", "perro", "cabra"]
```

Array

- Se puede crear arreglos con un tamaño y valor por defecto
- Agregar dos arreglos a uno por medio del operador **+**

```
var edades = Array(repeating: 12, count: 3)
// edades de tipo [Int], con valores [12, 12, 12]

var masEdades = Array(repeating: 6, count: 2)

//Añadir dos arreglos en uno

var edadesLista = edades + masEdades

print(edadesLista)
```

```
Swift Ver. 3.0 (Release)
Platform: Linux (x86_64)


[12, 12, 12, 6, 6]
```

Array

Iterar un arreglo

```
6  for item in animales{
7      print(item)
8  }
9
10 for (index, valor) in animales.enumerated() {
11     print("Item \(index + 1): \(valor)")
12 }
```

```
Pez dorado
vaca
perro
cabra
Item 1: Pez dorado
Item 2: vaca
Item 3: perro
Item 4: cabra
```

Array

Modificar

1. Modificando una posición [index]
2. Modificando varias posiciones [indexa...indexb]

```
1  animales[1] = "gato"
2  print("vaca cambia a gato:  \(animales)")
3
4  animales[1...3] = ["gallina", "cerdito", "caballo"]
5  print("de la posicion  1 a la 3 cambian:  \(animales)")
```

```
["Pez dorado", "vaca", "perro", "cabra"]
vaca cambia a gato:  ["Pez dorado", "gato", "perro", "cabra"]
de la posicion  1 a la 3 cambian:  ["Pez dorado", "gallina", "cerdito", "caballo"]
```

Array

Borrar

1. Método **remove(at : index)**
2. Método **removeLast()** Elimina el elemento en el ultimo index

```
1 print(animales)
2
3 animales.remove(at:3)
4 print(animales)
```

```
["Pez dorado", "gallina", "cerdito", "caballo"]
["Pez dorado", "gallina", "cerdito"]
```

Diccionarios

Un diccionario guarda asociaciones entre llaves del mismo tipo y valores del mismo tipo en una coleccion sin orden definido.
Cada valor está asociado con una llave única, que actúa como un identificador para un valor. Dictionary<Key, Value>

```
2  var animal = ["nombre":"perro", "raza": "pincher","edad":2]
3  var dicciVacio = [String]()
4
5  //manejo dicionarios
6  var keys = Array(animal.values)
7  var values = Array(animal.keys)
8
9  //sobreescribir
10 animal["nombre"] = "gato"
11 animal.removeValueForKey("edad")
12
13 //recorrer diccionarios
14 for(keys,values) in animal{
15     print ("llave:\(keys)valor:\(values)")
16 }
```

Diccionarios

- Inicializar un diccionario vacío

```
1 var nombresAnimales = [Int: String]()
```

Añadir

1. Se añaden elementos mediante **miDiccionario[key] = valor**

```
3 nombresAnimales[16] = "serpiente"
4 // contiene 1 key-value
5
6 nombresAnimales = [:]
7 //de nuevo vacio
```

Diccionarios

Modificar

1. De la misma manera que se agrega se sobreescribe
   miDiccionario[key] = valor
2. Método **updateValue(_, forKey:)**

```
8   nombresAnimales[2] = "pajaro"
9   nombresAnimales[5] = "conejo"
10  nombresAnimales[3] = "raton"
11  print(nombresAnimales)
12
13  nombresAnimales.updateValue("marmota", forKey : 5)
14  print("Actualizar key 5: ")
15  print(nombresAnimales)
```

```
[2: "pajaro", 5: "conejo", 3: "raton"]
Actualizar key 5:
[2: "pajaro", 5: "marmota", 3: "raton"]
```

Diccionarios

Borrar

1. Asignando un valor a **nil**
2. Método **removeValue(forKey:)**

```
17  print(nombresAnimales)
18
19  print("Borrar key 2: ")
20  nombresAnimales.removeValue(forKey: 2)
21  print(nombresAnimales)
22
23  print("Asignar nil a key 5: ")
24  nombresAnimales[5] = nil
25  print(nombresAnimales)
```

```
[2: "pajaro", 5: "marmota", 3: "raton"]
Borrar key 2:
[5: "marmota", 3: "raton"]
Asignar nil a key 5:
[3: "raton"]
```

Set

Es una colección desordenada de objetos distintos, se diferencia de un arreglo porque esta es una colección ordenada y de un diccionario , porque en él se encuentran desordenados los valores de referencia de claves específicos

Set<Element>.

- Inicialización:

```
2  var someSet = Set <String>()
3  let abcSet: Set = ["A","b","c"]
4  var foodSet = Set(["salad","chips","sandwiches"])
5
```

Set

Añadir, eliminar elementos:

```
7  foodSet.remove("Chips")
8  foodSet.insert("Soup")
9  foodSet.removeAll()
```

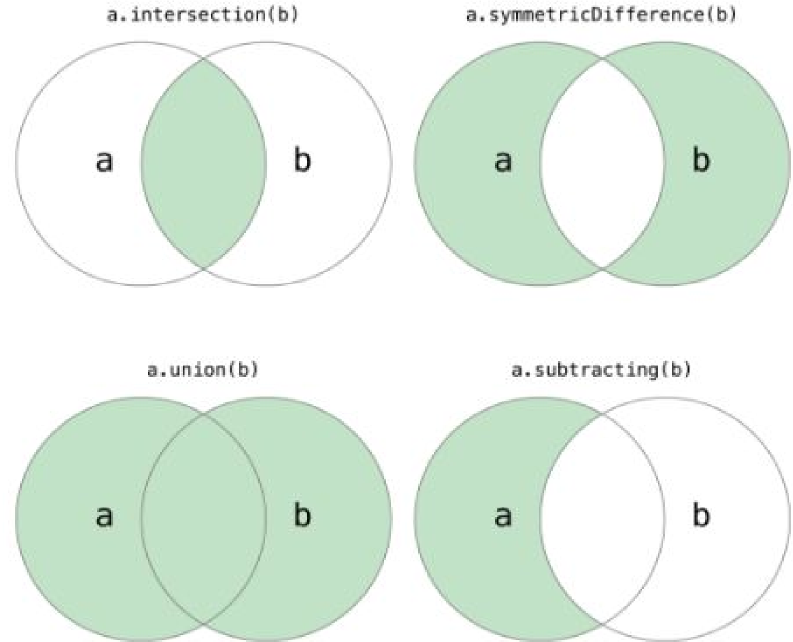
Métodos :

```
8  print (foodSet.contains("Salad"))
```

Set

Operaciones fundamentales:

- intersection(anotherSet)
- symetricDifference(anotherSet)
- union(anotherSet)
- subtracting(anotherSet)



Set

```
var letras : Set<Character> = ["a", "b", "c", "d", "e", "f"]
var vocales : Set<Character> = ["a", "e", "i", "o", "u"]

print("Union:")
var s = letras.union(vocales).sorted()
print(s)

print("Diferencia simetrica")
var d = letras.symmetricDifference(vocales).sorted()
print(d)
```

```
Union:
["a", "b", "c", "d", "e", "f", "i", "o", "u"]
Diferencia simetrica
["b", "c", "d", "f", "i", "o", "u"]
```

Set

Parentesco o igualdad:

- (==) Si dos sets contienen los mismos valores
- isSubset(of:) Todos los valores estan contenidos en otro
- isSuperset(of:) Contiene todos los valores de otro set
- isStrictSubset(of:) o isStrictSuperset(of:) Determinar si es subconjunto o superconjunto pero no igual.
- isDisjoint(with:) Determina si dos conjuntos tienen cualquiera de los valores en común.

Set

Parentesco o igualdad:

```
let animalesDomesticos: Set = ["🐶", "🐱"]
let animalesGranja: Set = ["🐮", "🐔", "🐑", "🐶", "🐱"]
let animalesCiudad: Set = ["🐦", "🐭"]

var a = animalesDomesticos.isSubset(of: animalesGranja)
print(a)
// true
var b = animalesGranja.isSuperset(of: animalesDomesticos)
// true
print(b)
var c = animalesGranja.isDisjoint(with: animalesCiudad)
//true
print(c)
```

```
true
true
true
```

CONTROLES DE FLUJO

- while
- if - else
- switch
- for
- for - in
- repeat while
- break
- continue

IF -ELSE

```
2  var peso = 8
3  if peso < 10{
4      if peso>5{
5          print("el peso esta entre 5 y 10 ")
6      }
7  }else{
8      print("mayor a 10")
9  }
```

```
5  var temperatura = 13
6  print(temperatura > 28 ? "Tomate una cerveza" : "Tomate un cafe")
7
8
```

- uso de if anidados para evaluar una condición
- ? :actúa como un if
- : actua como else

SWITCH

```
2   var animal = "A"
3
4   switch animal {
5
6       case "A":
7           print("es anfibio")
8
9       case "B":
10          print("es reptil")
11
12  default:
13          print("ninguno")
14  }
15
16  print (animal)
```

break
continue

SWITCH

```
23  var peso = 7
24  switch peso {
25
26      case 1:
27          print("esta en  a")
28
29      case 2...5:
30          print("esta en b")
31
32      case 6:
33          print("esta en  c")
34  default:
35          print("ninguno")
36  }
37  print (peso)
```

El switch en swift a diferencia de otros lenguajes contiene la condición para validar rangos específicos con tres puntos suspensivos

SWITCH

el switch en swift a diferencia de otros lenguajes contiene asociación de patrones como :

- asociación de intervalo
- cláusula where para comprobar condiciones adicionales
- asociación de tuplas

```
1  let animal = " vaca mamifero"
2  switch animal{
3      case "perro":
4          print("es mamifero")
5      case "lagartija":
6          print("es anfibio")
7      //con patrones
8      case let x where x.hasSuffix("mamifero"):
9          print("se ha encontrado vaca")
10     default:
11         print ("ninguno")
12 }
```

FOR IN

```
var animales = ["vaca","pollo","perro"]
//For each
for nombre  in animales{
    print(nombre)
}
//for enumeracion

for(nombre,animal) in animales.enumerate(){
    print("Nombre:\(nombre) animal \(animal)")
}
```

```
vaca
pollo
perro
Nombre:0 animal vaca
Nombre:1 animal pollo
Nombre:2 animal perro
iSwift:~ $
```

la palabra in basicamente es clave para realizar un ciclo sobre la colección de datos

WHILE

```
3   var animales = 1
4   while animales <= 100{
5       print (animales++)
6    }
```

se ejecutará solo si la condición se cumple

REPEAT WHILE

---

el ciclo do while se repetirá al menos una vez ya que primero ejecuta las lineas de codigo y despues verifica si la condición es verdadera o no

```
10  var peso = 0
11  repeat{
12      print("weight\(peso)")
13      peso++
14  }while peso <= 100
```

FUNCIONES

Se usa la palabra reservada func y el simbolo -> para devolver el tipo de función separando los argumentos.

```
func animal(nombre: String){
    print ("Nombre \(nombre)")
}
var perro = "aira"
animal(perro)

//funcion retorno
func sumar(a: Int , b : Int) -> Int{
return a+b
}
let resultado = sumar(10,b: 10)
print (resultado)
```

FUNCIONES

Las funciones en swift pueden recibir como parámetro otra función.

```
2  func sumatorio(desde a: Int, hasta b: Int, func f: (Int) -> Int) -> Int {
3      if a > b {
4          return 0
5      } else {
6          return f(a) + sumatorio(desde: a + 1, hasta: b, func: f)
7      }
8  }
9
10 func identidad(x: Int) -> Int {
11     return x
12 }
13
14 func doble(x: Int) -> Int {
15     return x + x
16 }
17
18 func cuadrado(x: Int) -> Int {
19      return x * x
20 }
```

```
55
110
385
```

PARÁMETROS CONSTANTES Y VARIABLES

```
func nombre (var a : String) -> String{
        a = a.uppercaseString
        print(a)
}
```

*parámetros no mutables
*var

FUNCIONES COMO TIPOS

```
func swapTwoInts(a: inout Int,b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
```

modificar el valor del parámetro: inout
valor que se pasa a la función
se modifica por la función
se pasa de nuevo a la función -reemplazar

Closures

- Equivalentes a expresiones lambda
- usadas para simplificar sintaxis

```
1  let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
2
3  let reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
4      return s1 > s2
5  })
```

CASTING DE TIPOS

Es una manera de comprobar el tipo de una instancia, o para tratar esa instancia como una superclase o subclase.

Se implementa con los operadores **is** y **as**.

- **is**: Comprueba si una instancia es un tipo de una subclase. Retorna true o false.
- **as?**: Hace el downcasting a un cierto tipo de clase si este falla retorna **nil**.
- **as!** : Intenta hacer el downcasting si este falla, un error de ejecucion es lanzado.

CLASES

Las clases en Swift son bloques de construcción de construcciones flexibles. Swift proporciona la funcionalidad que mientras las clases se declaran los usuarios no tienen que crear interfaces o archivos de implementación. Permite crear clases como un solo archivo y las interfaces externas se crean por defecto una vez que las clases se inicializan.

```
1  class Resolucion {
2      var ancho = 0
3      var alto = 0
4  }
5  class Video {
6      var resolucion = Resolucion()
7      var entrelazado = false
8      var frameRate = 0.0
9      var nombre: String?
10 }
```

CLASES

Beneficios:

- La herencia permite que una clase herede características de otra
- La conversión de tipos permite comprobar e interpretar el tipo de una instancia de clase en tiempo de ejecución.
- El conteo de referencias permite más de una referencia a una instancia de clase.

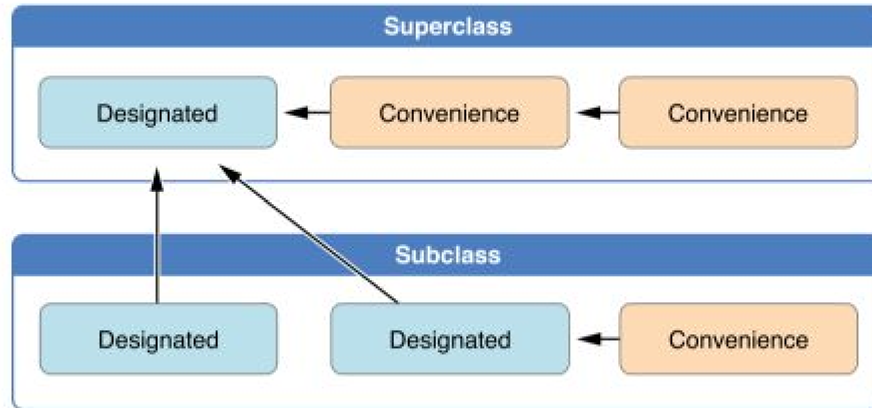
CLASES

Se puede **acceder a las propiedades** de una instancia utilizando la sintaxis "punto". En la sintaxis punto, se escribe el nombre de la propiedad inmediatamente después del nombre de la instancia, separados por un punto, sin ningún espacio (.):

```
let algunaResolucion = Resolucion()
let algunVideo = Video()

print("el ancho de algunaResolucion es \(algunaResolucion.ancho)")
// Prints "El ancho de algunaResolucion es 0"


print("El ancho de algunVideo es \(algunVideo.resolucion.ancho)")
// Prints "El ancho de algunVideo es 0"

//Puede asignarle un valor:
algunVideo.resolucion.ancho = 1280
```

Initializers

- Inicalizadores (Convenience)



Initializers

```swift
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}

struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

ESTRUCTURAS

Swift proporciona un bloque de construcción flexible para hacer uso de construcciones como las estructuras. Al hacer uso de estas estructuras se puede definir la construcción de los métodos y propiedades.

Similitudes de clases y estructuras:

- Definir propiedades para almacenar valores
- Definir los métodos para proporcionar funcionalidad
- Definir subscripts para proporcionar acceso a sus valores

Similitudes de clases y estructuras:

- Definir inicializadores para establecer su estado inicial
- Ampliar su funcionalidad
- Conforme a los protocolos estándar proporciona la funcionalidad de un cierto tipo

```
1 class AlgunaClase {
2     // definicion de clase
3 }
4 struct AlgunaEstructura {
5     // definicion de estructura
6 }
```

la inicialización de instancias en clases y estructuras es diferente.

```
1  struct Carac {
2      var peso = 0
3      var tamaño = 0
4  }
5  class Animal {
6      var carac = Carac()
7      var enfermedad = false
8      var nombre: String?
9  }
10 //inicializacion de instancias de clases y estructuras
11 
12 let unacar = Carac(peso: 50 , tamaño:30)
13 let unanimal = Animal()
14 //acceso a sus propiedades
15 print("El peso del animal es\(unacar.peso)")
16 print ("El peso del animal es\(unanimal.carac.peso)")
17 
```

HERENCIA

La herencia es una conducta fundamental que diferencia las clases de otros tipos en Swift , Cuando una clase hereda de otra, la clase que hereda se denomina subclase y la clase de la que hereda se denomina superclase .

para declarar una sub-clase se utiliza la siguiente sintaxis:

```
class Animal{
    var peso = 0
    var velocidad = 0
    var descripcion : String{
        return "viaja a \(velocidad)"
    }
    func correr(){
    }
}
let unanimal = Animal ()
print("Animal:\(unanimal.descripcion)")
//definicion de subclase
class Perro: Animal{
    var parasitos = true
}

let perro = Perro()
print ("El perro tiene\(perro.parasitos)")
perro.velocidad = 20
print("el perro \(perro.descripcion)")
```

```
Animal:viaja a 0
El perro tienetrue
el perro viaja a 20
iSwift:~ $
```

Overriding:

Propiedades:

```
1  class Car: Vehicle {
2      var gear = 1
3      override var description: String {
4          return super.description + " in gear \(gear)"
5      }
6  }
```

funciones:

```
1  class Train: Vehicle {
2      override func makeNoise() {
3          print("Choo Choo")
4      }
5  }
```

GENERICS

Se llama genéricos a **la posibilidad de pasar como un parámetro un tipo en lugar de un valor**.

Permite escribir, funciones flexibles reutilizables y tipos que pueden trabajar con cualquier tipo. Se puede escribir código que evite la duplicación, expresa su intención de una manera clara y abstraída.

Los tipos Array y Dictionary son colecciones genericas.

GENERICS

```swift
func swapTwoInts(inout a: Int, inout b: Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}


struct IntStack {
    var items = [Int]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

```swift
func swapTwoValues<T>(inout a: T, inout b: T) {
    let temporaryA = a
    a = b
    b = temporaryA
}


struct Stack<Element> {
    var items = [Element]()
    mutating func push(item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

CONTROL DE ACCESO

Public : Garantiza el acceso a entidades desde cualquier archivo que pertenece el módulo donde fueron definidas y desde cualquier otro que importe dicho módulo

Internal : Garantiza acceso únicamente al modelo origen

Private : Restringe totalmente el acceso a la entidad que lo implementa al archivo donde fue definida

```swift
public class SomePublicClass {                  // explicitly public class
    public var somePublicProperty = 0            // explicitly public class member
    var someInternalProperty = 0                 // implicitly internal class member
    fileprivate func someFilePrivateMethod() {}  // explicitly file-private class member
    private func somePrivateMethod() {}          // explicitly private class member
}

class SomeInternalClass {                       // implicitly internal class
    var someInternalProperty = 0                 // implicitly internal class member
    fileprivate func someFilePrivateMethod() {}  // explicitly file-private class member
    private func somePrivateMethod() {}          // explicitly private class member
}

fileprivate class SomeFilePrivateClass {        // explicitly file-private class
    func someFilePrivateMethod() {}              // implicitly file-private class member
    private func somePrivateMethod() {}          // explicitly private class member
}

private class SomePrivateClass {                // explicitly private class
    func somePrivateMethod() {}                  // implicitly private class member
}
```

PROTOCOLOS

- Similares a una Interfaz en Java
- Son un esquema de los métodos, propiedades y otros requerimientos que deben ser adoptados por una clase estructura o enum.
- *Set* y *Get* determinan el comportamiento.

```
1 protocol SomeProtocol {
2     var mustBeSettable: Int { get set }
3     var doesNotNeedToBeSettable: Int { get }
4 }
```

ARC

- Automatic Reference Counter - Motor encargado de manejar la memoria en swift.
- Si no hay referencias de instancia a una clase, ARC libera esta memoria
- Para evitar que se borre la instancia de una clase se debe asignar a una variable (referencia fuerte).
- Las propiedades de un objeto tienen referencia fuerte

Ejemplo básico- cálculo factorial

```
3  func factorial( numero: Int) -> Int {
4      if numero == 1 {
5          return 1
6      } else {
7          return numero * factorial(numero - 1)
8      }
9  }
10 print (factorial(3))
```

Ejemplo Intermedio - Merge Sort <Generics>

```
//mergesort

func elementsInRange<T>(a: [T], start: Int, end: Int) -> ([T]) {
    var result = [T]()

    for x in start...end {
        result.append(a[x])
    }

    return result
}

func merge<T: Comparable>(a: [T], b: [T], mergeInto acc: [T]) -> [T] {
    if a == [] {
        return acc + b
    } else if b == [] {
        return acc + a
    }

    if a[0] < b[0] {
        return merge(a: elementsInRange(a: a,start: 1, end: a.count), b: b, mergeInto: acc + [a[0]])
    } else {
        return merge(a: a,b: elementsInRange(a: b,start: 1, end: b.count), mergeInto: acc + [b[0]])
    }
}

func mergesort<T: Comparable>(a: [T]) -> [T] {
    if a.count <= 1 {
        return a
    } else {
        let firstHalf = elementsInRange(a: a,start: 0,end: a.count/2)
        let secondHalf = elementsInRange(a: a,start: a.count/2,end: a.count)

        return merge(a: mergesort(a: firstHalf),b: mergesort(a: secondHalf), mergeInto: [])
    }
}
```

Ejemplo Avanzado - Segment Tree

```swift
public class SegmentTree<T> {

    private var value: T
    private var function: (T, T) -> T
    private var leftBound: Int
    private var rightBound: Int
    private var leftChild: SegmentTree<T>?
    private var rightChild: SegmentTree<T>?

    public init(array: [T], leftBound: Int, rightBound: Int, function: @escaping (T, T) -> T) {
        self.leftBound = leftBound
        self.rightBound = rightBound
        self.function = function

        if leftBound == rightBound {
            value = array[leftBound]
        } else {
            let middle = (leftBound + rightBound) / 2
            leftChild = SegmentTree<T>(array: array, leftBound: leftBound, rightBound: middle, function: function)
            rightChild = SegmentTree<T>(array: array, leftBound: middle+1, rightBound: rightBound, function: function)
            value = function(leftChild!.value, rightChild!.value)
        }
    }

    public convenience init(array: [T], function: @escaping (T, T) -> T) {
        self.init(array: array, leftBound: 0, rightBound: array.count-1, function: function)
    }

    public func query(withLeftBound: Int, rightBound: Int) -> T {
        if self.leftBound == leftBound && self.rightBound == rightBound {
            return self.value
        }

        guard let leftChild = leftChild else { fatalError("leftChild should not be nil") }
        guard let rightChild = rightChild else { fatalError("rightChild should not be nil") }

        if leftChild.rightBound < leftBound {
            return rightChild.query(withLeftBound: leftBound, rightBound: rightBound)
        } else if rightChild.leftBound > rightBound {
            return leftChild.query(withLeftBound: leftBound, rightBound: rightBound)
        } else {
            let leftResult = leftChild.query(withLeftBound: leftBound, rightBound: leftChild.rightBound)
            let rightResult = rightChild.query(withLeftBound:rightChild.leftBound, rightBound: rightBound)
            return function(leftResult, rightResult)
        }
    }

    public func replaceItem(at index: Int, withItem item: T) {
        if leftBound == rightBound {
            value = item
        } else if let leftChild = leftChild, let rightChild = rightChild {
            if leftChild.rightBound >= index {
                leftChild.replaceItem(at: index, withItem: item)
            } else {
                rightChild.replaceItem(at: index, withItem: item)
            }
            value = function(leftChild.value, rightChild.value)
        }
    }
}
```

Referencias

[1]https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/

[2]http://www.campusmvp.es/recursos/post/Comparando-valores-y-referencias-en-varios-lenguajes-de-programacion.aspx

[3]https://www.tutorialspoint.com/swift/swift_classes.htm