UNIVERSIDAD NACIONAL DE COLOMBIA
SEDE BOGOTÁ
FACULTAD DE INGENIERÍA

Lenguajes de programación
Taller Swift
2016 - 2

Tutorial Swift

Nicolás Larrañaga Cifuentes Liseth Briceño Albarracín
Angela María Muñoz Medina

30 de octubre de 2016

Índice

1. Instalación

Swift es soportado principalmente en el sistema operativo de OSX (dispositivos apple), también cuenta con un compilador en linux; sin embargo para windows no existen compiladores oficiales actualmente, por lo que es necesario utilizar un compilador en linea (desarrollado por IBM).

1.1. OSX

Al ser creado por Apple para desarrollar apps de iOS, Mac, Apple TV y Apple Watch, solo se necesita descargar un IDE para codificar, diseñar y probar. Para esto usaremos Xcode 8 que es la ultima version estable, cabe resaltar que versiones anteriores a Xcode 6.4 no soportan swift.

1.1.1. Xcode

El ultimo sistema operativo para Mac llamado MacOS 10.12 Sierra permite la facil instalacion de Xcode mediante los siguientes pasos:

1. Entrar a la App Store y buscar Xcode
2. Dar clic en obtener y luego en instalar (ver figura 1)
3. Esperar, Xcode tiene un tamaño de 4.63GB (ver figura 2)
4. Buscar la aplicacion instalada en Launchpad (ver figura 3)
5. Aceptar terminos e instalar Xcode (ver figura 4)

Figura 1:

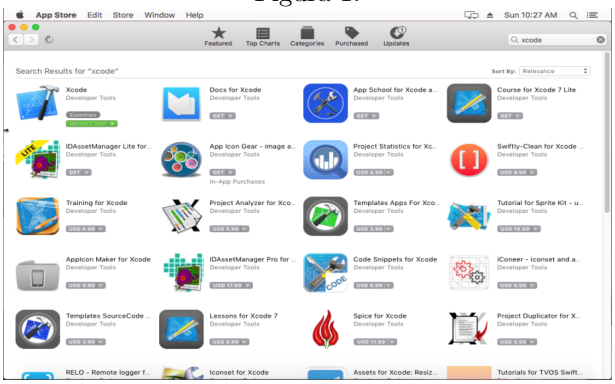
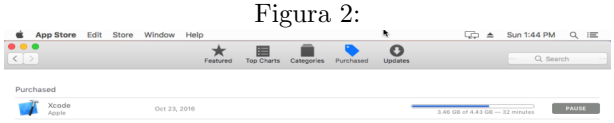


Figura 2:



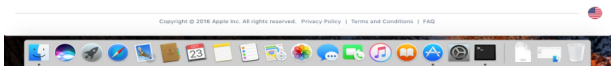
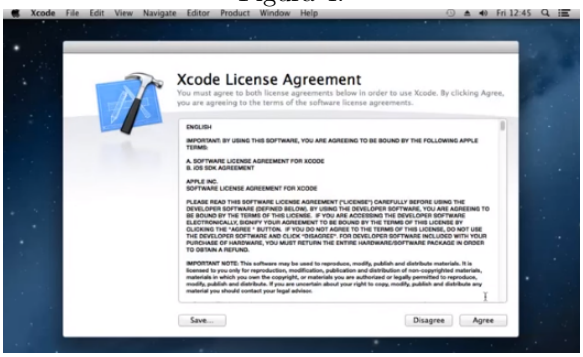


Figura 3:

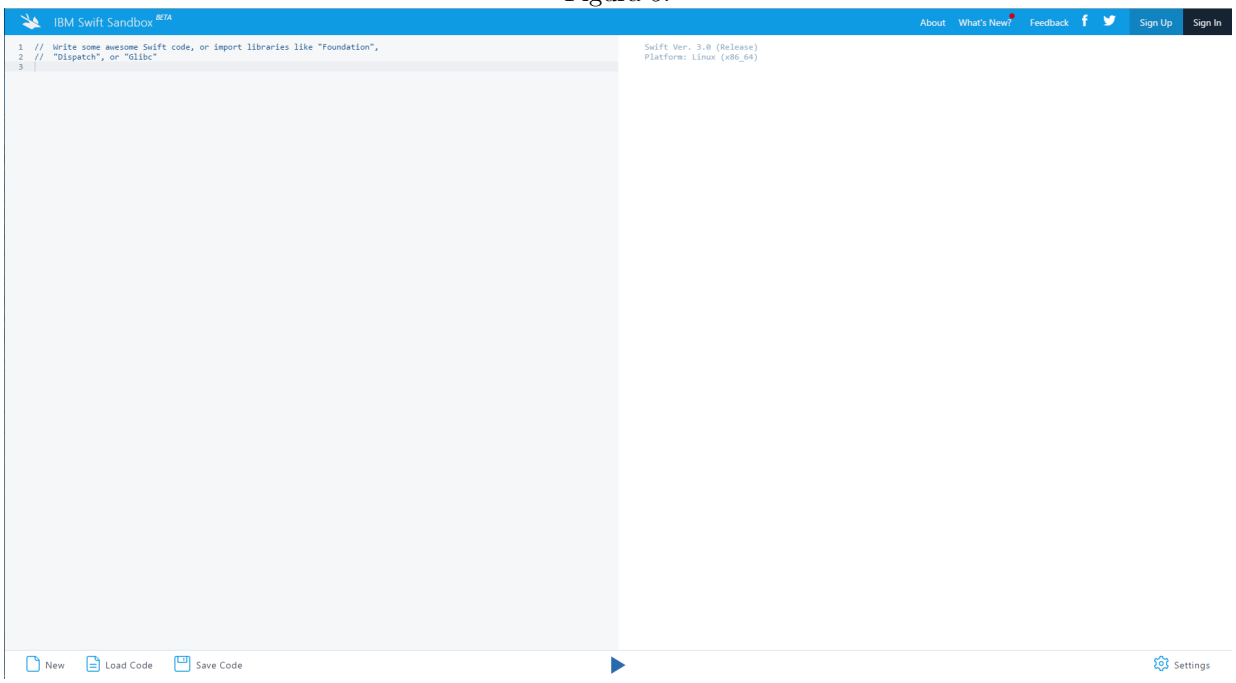
Figura 4:



1.2. Windows

Como fue descrito anteriormente, windows no cuenta con un compilador oficial de swift, sin embargo para los fines de este tutorial es suficiente utilizar un compilador (playground) en linea, desarrollado por IBM, para ello entrar a `https://swiftlang.ng.bluemix.net/#/repl`.

Figura 5:



2. Primeros pasos con swift

2.1. Operadores basicos

A continuación se presentan los operadores de los que dispone swift:

- Asignacion =
- Aritmeticos +,-,*,/
- Residuo %
- Incremento- Decremento ++,–
- Comparacion ==, !=,>,<,<=,>=
- Identidad == , !=== (tener en cuenta que ambos operadores son diferentes en su implementación, ver sección 9.3
- Rango a..¡b a..b
- Logicos !, &&, ||

2.2. Caracteres y String

La sintaxis para la manipulacion y creacion de cadenas es similar a C, la concatenacion de cadenas se realiza con el operador + y ademas de ello se encuentra la mutabilidad de la cadena que es administrado entre por la eleccion entre una constante , variables, literales y expresiones en cadenas mas largas,en un proceso denominado interpolacion de cadenas, lo cual facilita la visualizacion e impresión.

A pesar de la sintaxis, en Swift el tipo String es una aplicacion de cadena rapida y sencilla. Cada cadena compone de caractares Unicode independimiente de la codificacion y proporciona soporte para acceder a diversas representaciones Unicode

2.3. Literales

Un literal de cadena es una secuencia fija de caracteres de texto rodeado por un par de comillas dobles "" .

```
let string = "hello world"
```

2.3.1. Inicializacion de String vacio

Para inicializar un string vacio se asigna la cadena vacia () o el inicializador de tipo String (String())

```
var stringA  = ""
var stingB =  String()
```

2.3.2. Mutabilidad

El tipo de dato en Swift es mutable en Swift, lo que permite, realizar cambios sin necesidad de crear una nueva instancia de la clase o de tipo de dato.

```
 var variableString = "hola"
 variableString += "  mundo"
 // la variableString ahora es hola mundo
```

2.3.3. Tipo de valor

En Swift el tipo String es un tipo valor. Si se crea un nuevo String de valor,el String de valor se copia cuando se pasa a una función o metodo, o cuando se asigna una constante o una varible.

2.3.4. Trabajando con caractares

Las operaciones y propiedadas mencionadas anteriormente se aplican pra el dato de tipo Character

2.3.5. Interpolacion de String

La interpolación de cadenas es una manera de construir una nuevo valor de String a partir de una mezcla de constantes, variables, literales y expresiones mediante la inclusión de sus valores dentro de una cadena literal. Cada elemento que se inserta en la cadena literal se envuelve en un par de paréntesis, precedido por una barra invertida:

```
var num = 10
var cadenaInter = "El precio de\(numero)"
print (cadenaInter)
// imprime el precio de 10
```

2.3.6. Comparacion de String

Swift compara los valores canónicos de cada caracter unicode

```
 let primera = "cadena\u{E9}?"
 let segunda = "cadena\u{65}\u{301}?"
  if primera == segunda {
 print ( "las dos son consideradas igual" )
 }
 //imprime las dos son consideradas igual
```

3. Colecciones

3.1. Arrays

Un arreglo almacena valores del mismo tipo en una lista ordenada. El mismo valor puede aparecer en un arreglo varias veces en diferentes posiciones.
La inicializacion de un arreglo es : $Array< tipoElemento >$ que se puede resumir en $[\ tipoElemento\ ]$

3.1.1. Inicializacion de Array

Se puede inicializar un arreglo vacio:

```
var edades = [Int]()
```

O se puede inicializar con varios elementos:

```
var elementos: [String] = ["cloro", "magnesio", "aluminio"]
```

3.1.2. Metodo añadir, borrar, modificar

Para añadir elementos a un arreglo se tiene el metodo .apped() o se puede indicar una posicion con el método $insert(, at :)$.

```
elementos.append("azufre")
elementos.insert("neon", at : 1)
```

Para borrar un elemento se utilizan los metodos remove(at : index) y removeLast()

```
elementos.remove(at : 1)
elementos.removeLast()
```

Si se quiere modificar un elemento:

```
elementos[3] = "oro"
```

Para modificar varios elementos a la vez se puede utilizar rangos:

```
elementos[3..5] = ["oro", "litio", "cobalto"]
```

3.2. 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.
La inicializacion de un diccionario es : $Dictionary< Key, Value >$

3.2.1. Inicializacion de un diccionario

Se puede inicializar un diccionario vacio:

```
var nombredeNumeros = [Int: String]()
```

3.2.2. Metodo añadir, borrar, modificar

Se añaden elementos mediante miDiccionario[key] = valor

```
nombredeNumeros[2] = "dos"
// nombredeNumeros contiene 1 par key-value
nombredeNumeros = [:]
// nombredeNumeros de nuevo es un diccionario vacio [Int: String]
```

Para borrar un valor se puede asignar nil a una llave o usando el método removeValue(forKey:)

```
nombredeNumeros[2] = "dos"
nombredeNumeros[10] = "diez"
nombredeNumeros[12] = "doce"
nombredeNumeros[1] = "uno"
//[dos, diez, doce, uno]

nombredeNumeros[2] = nil
//[diez, doce, uno]

nombredeNumeros.removeValue(forKey: 1)
//[diez, doce]
```

Una opción para modificar el valor de una llave es sobreescribir de la misma manera que se agrega: miDiccionario[key] = valor, o por el método $updateValue(\_, forKey:)$

```
nombredeNumeros[12] = "doceInt"
//[diez, doceInt]
nombredeNumeros.updateValue("doce", forKey: 12)
//[diez, doce]
```

3.3. Set

Los sets son una coleccion desordenada de objetos, su principal uso es el de decidir si un objeto pertenece o no a un conjunto eficientemente y no interesa el orden en dicho conjunto, su implementación hace uso del concepto de *HashTable* La inicialización de un set es : $Set< Element >$. En Swift la mayoría de los tipos en la librería estandar poseen un metodo de hash, lo cual hace posible la existencia de sets que contengan enteros, booleanos, strings e incluso otros sets.

3.3.1. Inicializacion de un Set

Se puede inicializar un set vacio:

```
var letras = Set<Character>()
print("La cantidad de letras es \(letras.count).")
// imprimira que tiene 0 letras
```

O de la siguiente manera con valores:

```
var vocales = Set(["a","e","i","o","u"])
```

3.3.2. Hash Values para tipos Set

Un tipo debe ser hashable con el fin de ser almacenado en un conjunto, es decir, el tipo debe proporcionar una forma de calcular un valor hash por sí mismo. Un valor hash es un valor entero que es el mismo para todos los objetos que comparan igualmente, de manera que si a == b, se deduce que a.hashValue == b.hashValue

Todos los tipos básicos de Swift (como String, int, double, y Bool) son hashable por defecto, y se pueden utilizar como tipos de valores en un Set o como tipo de llaves y valores en un Diccionario. Los valores de enumeración tambien son hashable.

3.3.3. Metodo añadir, borrar, modificar

Se añaden elementos mediante el método insert(elemento)

```
letras.insert("a")
letras.insert("b")
letras.insert("c")
```

Para borrar un valor se utiliza el método remove(elemento), si se quiere vaciar el conjunto se utiliza .removeAll()

```
letras.remove("c")
letras.removeAll()
```

3.3.4. Metodos y operaciones fundamentales

Si se quiere encontrar un elemento especifico se puede utilizar el metodo .contains(elemento)
Los Set tienen las siguientes operaciones fundamentales:

```
letras.intersection(vocales)
letras.symetricDifference(vocales)
letras.union(vocales)
letras.subtracting(vocales)
```

3.3.5. Parentesco o igualdad

```
(letras==vocales)...
//Si dos sets contienen los mismos valores

letras.isSubset(of: vocales )
//Todos los valores estan contenidos en otro

letras.isSuperset(of: vocales)
//Contiene todos los valores de otro set

letras.isStrictSubset(of: vocales)
letras.isStrictSuperset(of: vocales)
//Determinar si es subconjunto o superconjunto pero no igual.

letras.isDisjoint(with:vocales)
//Determina si dos conjuntos tienen cualquiera de los valores en común.
```

4. Control de Flujo

Swift ofrece una variedad de estados de flujo de control. Estos incluyen while los bucles para realizar una tarea varias veces; if , guard , y switch ; y las declaraciones tales como break y continue para transferir el flujo de ejecución a otro punto en el código. Swift también proporciona un for - in bucle que hace que sea fácil de iterar sobre matrices, diccionarios, rangos, cadenas y otras secuencias.

4.1. For in

El for in es un ciclo que se usa para repetir una secuencia varias veces, este contiene rangos de números, elementos de una matriz, o caracteres de una cadena.

Este ejemplo imprime las primeras entradas en la tabla seis veces:

```
for index in 1...7 {
    print("\(index) times 7 is \(index * 7)")
}
// 1 times 7 is 7
```

```
// 2 times 7 is 14
// 3 times 7 is 21
// 4 times 7 is 28
// 5 times 7 is 49
```

Se hace uso de un for in para iterar sobre arreglos como:

```
let nombres = [ "Anna" , "Alex" , "Brian" , "Jack" ]
for nombre in nombres {
 print ( "Hello, \( nombre ) !" )
}
// Hello, Anna!
// Hello, Alex!
// Hello, Brian!
// Hello, Jack!
```

También se pueden recorrer diccionarios como :

```
let numeroEdad = [ "spider" : 8 , "ant" : 6 , "cat" : 4 ]
 for ( animalNombre , legEdad ) in numeroEdad {
 print ( " \( animalNombre ) s have \( legEdad ) edad" )
 }
 // ants have 6 edad
 // spiders have 8 edad
 // cats have 4 edad
```

4.2. For

La estructura for es similar al lenguaje C, pero Swift permite escribirla sin parentesis de la siguiente manera:

```
for var  tamaño=0; tamaño <30 ; tamaño++{
        print (tamaño)
```

4.3. while

Un ciclo while realiza un conjunto de declaraciones hasta que una condición se convierte en falsa . Este tipo de bucles se usan mejor cuando no se conoce el número de iteraciones antes de que comience la primera iteración. Swift ofrece dos tipos de ciclos de while:

- while evalúa su estado en el inicio de cada pasada a través del ciclo.
- repeat - while evalúa su condición al final de cada paso a través del ciclo.

Ejemplo while:

```
var animales = 1
while animales <= 100{
    print (animales++)
}
```

4.4. Do- while

La otra variación del ciclo while, conocida como la repeat - while bucle, realiza una sola pasada a través del bloque del ciclo primero, antes de considerar la condición . A continuación, sigue repitiendo el ciclo hasta que la condición sea falsa.

```
var peso = 0
repeat{
    print("weight\(peso)")
    peso++
}while peso <= 100
```

4.5. If

El if la declaración tiene condiciones que se ejecutan sólo si la condición es verdadero.

```
if numero < 10{
    if numero>5{
        print("el numero es menor a 10 y mayor a 5")
    }
}else{
    print("mayor a 10")
}
```

4.6. Switch

la declaración switch compara un valor contra uno o más valores del mismo tipo.
Una caracteristica en la que se diferencia el switch entre C y Swift es el "fallthrough". es que el switch de Swift encuentra un caso verdadero, lo ejecuta y termina. Mientras que En C, el switch evalua las condiciones en orden de aparicion, y aun cuando encuentra una verdadera, continua evaluando el resto de expresiones , es por ello que se coloca break explicitamente. En swift, el break no es necesario.

```
switch cadena {

    case "A":
        print("es a")

    case "B":
        print("es b")

default:
        print("ninguno")
}
```

Ademas de ello Swift cuenta con dos usos adicionales como los siguientes:

- Switch con rangos:

```
switch cadena {
    case 1:
        print("es a")
    case 2...5:
        print("es b")
    case 6:
        print("es b")
default:
        print("ninguno")
}
```

- Switch con patrones:

```
let nombre = " roberto lujan"
switch nombre{
        case "alberto":
        print("alberto")
    case "juan":
        print("juan")
   //con patrones
    case let x where x.hasSuffix("lujan"):
        print("encontramos a roberto")
    default:
        print ("ninguno")
}
```

4.7. Tuplas

Swift añade una funcionalidad para juegos y abtraccion de planos. Se pueden utilizar tuplas con las estructuras switch para encontrar cuadrantes,patrones, o varios valores exactos. A continuación se muestra un ejemplo:

```
switch Tupla{
        case(0,0,0)
                print (1)
        case(_,0,_)
                print (2)
    default:
                print(3)
}
```

En el primer caso de la estructura switch, se busca que 3 elementos sea exactamente uno (0,0,0). En el segundo caso, se utiliza el simbolo para señalar que se desea ignorar ese caracter. Esto significa que busca solo una condicion, la cual es el segundo elemento.Para el primero y el tercero, cualquier caracter es valido.

5. Funciones

Las funciones son bloques autónomos de código que realizan una tarea específica. se debe asignar un nombre a la funcion , y este sera usado para llamar alguna tarea especifica.

Los parámetros pueden proporcionar valores predeterminados para simplificar las llamadas a funciones y se pueden pasar como parámetros de salida, que modifican una variable, pasada una vez que la función ha terminado su ejecución.

Las funciones en Swift tienen un tipo. Por ejemplo: String es un tipo. Integer, Character, Boolean. Una función, al declararse, tiene un tipo, que consiste en los tipos de los parametros y el tipo de retorno. Lo anterior permite utilizar las funciones como parametros en llamados de metodos, o inclusive asignar funciones a variables declaradas por el usuario.

5.1. Definicion y llamado de funciones

Cuando se define una funcion, puede definir opcionalmente uno o mas parametros Para hacer uso de la funcion se llama con su nombre y se pasa los valores de entrada conocido como argumentos que coinciden con los tipos de parametros de la funcion. Los argumentos de una función se deben proporcionar en el mismo orden que la lista de los parametros de la función.

La funcion que se muestra a continución muestra como la funcion utiliza la palabra reservada func y utiliza $->$ que indica el tipo de la función de retorno

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

5.2. Funcion con parametros

Las funciones pueden tener varios parámetros de entrada, que se escriben entre paréntesis de la función, separados por comas. Ej.

```
func greetAgain(person: String) -> String {
    return "Hello again, " + person + "!"
}
print(greetAgain(person: "Anna"))
// Prints "Hello again, Anna!"
```

5.3. Funcion sin parametros

Las funciones que no son necesarios para definir los parámetros de entrada.Acontinuacion se muestra un ejemplo que imprime un valor.

```
func decirHola() {
    print("hola!")
}
decirHola()
```

5.4. Funcion que no retorna valor

```
func greet(persona: String) {
    print("Hola, \(persona)!")
}
greet(persona: "David")
// Prints "Hola, David!"
```

Debido a que no tiene que devolver un valor, la definición de la función no incluye la flecha de retorno $->$ o un tipo de retorno.

5.5. Funcion que retorna valor

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var cMin = array[0]
    var cMax = array[0]
    for valor in array[1..<array.count] {
        if valor < cMin {
            cMin = valor
        } else if valor > cMax {
            cMax = valor
        }
    }
    return (cMin, cMax)
}

let prueba = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(prueba.min) and max is \(prueba.max)")
// Prints "min is -6 and max is 109"
```

5.6. Nombres internos y externos

Las funciones en Swift pueden nombrar sus parametros internamente y externamente. Esto quiere decir que cuando un parametro es llamado desde el alcance de la funcion, se utiliza el nombre interno. Cuando la funcion es utilizada por fuera de su alcance (cuando es llamada), se utilizan los nombres externos.

5.7. parametros constantes y variables

Los parametros de Swift se manejan como constantes por defectos. En una funcion , no se ha modificado los paramtro parametros. Esto es porque esos parametros se comportan como constantes y no son mutables. Para solucionar esto, hay que agregar la palabra var antes del identificador de la variable para indicar que esa variable es mutable. Al hacer esto, se genera una copia local del parametro, el cual puede ser modificado. Es importante resaltar que cuando el programa salga del alcance de la funcion, los cambios realizados en la variable local seran ignorados.

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

5.8. Funciones como tipos

Los parámetros de la función son constantes por defecto. Tratar de cambiar el valor de un parámetro de función desde dentro del cuerpo de esa función da como resultado un error en tiempo de compilación. Esto significa que no se puede cambiar el valor de un parámetro por error. Si desea una función para modificar el valor de un parámetro, y desea que estos cambios persisten después de la llamada a la función ha terminado, defina este parámetro como un parámetro de salida en su lugar.

Se escribe un parametro de entrada-salida colocando el inout derecho de palabra clave antes de tipo de un parámetro. Un parámetro de entrada-salida tiene un valor que se pasa a la función, se modifica por la función, y se pasa de nuevo fuera de la función de reemplazar el valor original. Ejemplo:

```
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)")
```

6. Clases y estructuras

Las clases y estructuras son construcciones flexibles multiproposito que se convierten en los bloques de construccion de cualquier programa. A diferencia de otros lenguages de programación, en swift no se requiere la creación de nuevos archivos e interfaces para la declaración de una nueva clase o estructura.

En su funcionalidad las clases y las estructuras se asemejan bastante, como por ejemplo en los siguientes aspectos

- Definición de propiedades para guardar variables
- Definición de metodos para proveer de una funcionalidad
- Definición de inicializadores para establecer su estado inicial.

6.1. Clases

Como es de esperar en un lenguage que soporte el paradigma de la programación orientada a objetos, swift cuenta con una manera de declarar, instanciar y abstraer 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.
- De Inicializar permite una instancia de una clase para liberar cualquier recurso que se ha asignado.
- El conteo de referencias permite más de una referencia a una instancia de clase.

6.1.1. Declaración e instanciación

Se introduce clases con la palabra clave *class*

```
class Video {
    var resolucion = Resolucion()
    var entrelazado = false
    var frameRate = 0.0
    var nombre: String?
}
//Se describe una estructura para uso posterior
struct Resolucion {
    var ancho = 0
    var alto = 0
}
```

En el ejemplo anterior La resolución, se inicia con una nueva instancia de estructura Resolución, que infiere las propiedades del tipo Resolución
La sintaxis para la creación de instancias es muy similar para estructuras y clases:

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

6.1.2. Propiedades de acceso

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 (.):

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

Puede profundizar en sub-propiedades, como la propiedad de ancho en la propiedad de resolución de un vídeo:

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

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

Las clases son tipos de referencia
A diferencia de los tipos de valores, los tipos de referencia no se copian cuando se les asigna a una variable o constante, o cuando se pasan a una función. En lugar de una copia, hace una referencia a la misma instancia existente.

```
let mov = Video()
mov.resolucion = hd
mov.entrelazado = true
mov.nombre = "1080i"
mov.frameRate = 25.0
```

6.1.3. Los operadores de identidad

Dado que las clases son tipos de referencia, es posible que varias constantes y variables se refieran a la misma y única instancia de una clase detrás de escenas.
A veces puede ser útil para averiguar si dos constantes o variables se refieren exactamente a la misma instancia de una clase. Para permitir esto, Swift ofrece dos operadores de identidad:

- Idéntico a (===)
- No idéntico a (!==)

Se debe tener en cuenta que ïdéntico a"(representado por tres signos de igual o ===) no significa lo mismo que ïgual"(representada por dos signos de igual, o ==):

Ïdéntico a"("Identical to") significa que dos constantes o variables de tipo clase se refieren exactamente a la misma instancia de clase. Ïgual a"("Equal to") significa que dos instancias son considerados ïguales.º .eqquivalente.en valor, desde un significado apropiado de ïguales", como se define por el diseño del tipo.

6.1.4. Control de acceso

El concepto de control de acceso de Swift está basado en la idea de módulos y source files. Un módulo es una unidad de código la cual es importada mediante la palabra reservada *import*.

Los niveles de acceso en swift son los siguientes

- acceso público: estas entidades pueden ser usadas dentro del mismo modulo o fuera de él
- acceso interno: estas entidades pueden ser usadas únicamente dentro del framework en que fue definido.
- acceso file-private: similar al acceso interno, restringe el uso al módulo (o archivo de codigo fuente) en el que está definido.
- acceso privado: restringe el acceso unicamente al contexto léxico de su declaración. (i.e la clase donde se declaró, la estructura, etc).

Las palabras reservadas específicadas para hacer uso del control de acceso se muestra a continuación.

```
public class SomePublicClass {}
internal class SomeInternalClass {}
fileprivate class SomeFilePrivateClass {}
private class SomePrivateClass {}
```

6.1.5. Herencia

Una clase puede heredar métodos, propiedades y otras características de otra clase. Cuando una clase hereda de otra, la clase que hereda se conoce como una subclase, y la clase que hereda de que se conoce como su superclase.
La herencia es un comportamiento fundamental que diferencia una clase de otros tipos en Swift.
Las clases en Swift pueden llamar y acceder a los métodos, las propiedades y los subíndices que pertenecen a su superclase y pueden proporcionar sus propias versiones (overriding) de esos métodos y las propiedades para modificar su comportamiento.

Sintaxis de una subclase:

```
class Subclase: Superclase {
    // subclase deifinicion
}
```

Ejemplo:

```
class Animal {
    var nombre : String?
    var edad = 0
    var descripcion: String {
        return "\(nombre) tiene \(edad) años"
    }
    func hablar() {
        // ...
    }
}

let algunAnimal = Animal()

class Perro: Animal {
    var raza : String?
}
let perro = Perro()
perro.raza = "Pastor del caucaso"

perro.edad = 2
```

Overriding:
Una subclase puede proporcionar su propia implementación personalizada de la instancia de un método , el tipo de método y tipo de propiedad que de otro modo hereda de una superclase.
Se puede acceder a la versión de la superclase de un método, una propiedad o subscript utilizando el prefijo *super*:

```
class Gato: Animal {
    override func hacer() {
        print("Miau")
    }
}

let gato = Gato()
gato.hacer()
// imprime "Miau"

class Raton: Animal {
    var meses = 5
    override var descripcion: String {
        return super.descripcion + " y \(meses)"
    }
}
```

6.2. Estructuras

Como se dijo anteriormente la sintaxis de una estructura es muy similar a la de una clase:

```
struct persona {
    var edad : Int
```

```
    var nombre: String
    var sexo: String
}
```

Todas las estructuras tienen un inicializador miembro por miembro generado automáticamente, que se puede utilizar para inicializar las propiedades de los miembros de las nuevas instancias de estructura. los valores iniciales de las propiedades de la nueva instancia se pueden pasar a la inicializador miembro por miembro por su nombre:

```
let vga = Resolucion(ancho: 640, alto: 480)
```

Las estructuras son tipos de valor
Un tipo de valor es un tipo cuyo valor se copia cuando se asigna a una variable o constante, o cuando se pasa a una función.

Pautas para pensar utilizar estructuras:

- El propósito principal de la estructura es encapsular un par de valores de datos relativamente simples.
- Todas las propiedades almacenadas por la estructura son en sí mismos tipos de valores, que también se esperaría que se va a copiar en lugar de referencia.
- La estructura no necesita heredar propiedades o comportamiento de otro tipo existente.

7. Tipos de valor y de referencia

7.1. Tipos de 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.

7.2. Tipos de 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

8. Características de Swift

8.1. Closures

Los closures son bloques de funcionalidad autónomos que se pueden pasar alrededor y utilizar en su código. Closures en Swift son similares a los bloques en C y Objective-C y a lambdas en otros lenguajes de programación.
Los cierres pueden capturar y almacenar referencias de cualquier constante y variable del contexto en el que se definen. Esto se conoce como el closure sobre esas constantes y variables. Swift se encarga de toda la gestión de memoria.

Funciones globales y anidadas, en realidad son casos especiales de los closures. Los cierres tienen una de tres formas:

- funciones globales son closures que tienen un nombre y no capturan ningun valore.
- Funciones anidadas son closures que tienen un nombre y pueden capturar los valores de la función que encierran.

- Expresiones de closure son cierres sin nombre escrito en una sintaxis ligera que puede capturar los valores de su contexto circundante.

Las *expresiones de closure* son una manera de escribir los closures en una línea breve, centrado en la sintaxis. Proporcionan varias optimizaciones de sintaxis para la escritura de los closures en una forma abreviada y sin pérdida de claridad o de intención.

Método Sorted

sorted(by :), ordena una matriz de valores de un tipo conocido, basándose en la salida de un cierre de clasificación que proporcione. Una vez se complete el proceso de clasificación, sorted(by :) devuelve una nueva matriz del mismo tipo y tamaño que la anterior, con sus elementos en el orden de clasificación correcta. La matriz original no se modifica por el método.

```
let nombres = ["Chris", "Alex", "Ewa", "Barry", "Daniela"]

func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
var reversedNames = nombres.sorted(by: backward)
// reversedNames = ["Ewa", "Daniela", "Chris", "Barry", "Alex"]
```

8.1.1. Sintaxis Closures

La sintaxis tiene esta forma general:

```
{ (parameters) -> return type in
    statements
}
```

Los parámetros de la sintaxis de las expresiones de cierre pueden ser parámetros in-out, pero no pueden tener un valor predeterminado. Las tuplas también se pueden utilizar como tipos de parámetros y tipos de devolución.

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

8.2. ARC

El ARC o Automatic Reference Counting es el motor que se encarga de manejar la memoria en swift, su función más relevante es la de liberar memoria en caso de que la instancia que la ocupa no sea necesaria, esto ocurre cuando una instancia no tiene una **referencia fuerte**.

Una referencia fuerte ocurre cuando una instancia es asignada a una variable o una constante, también las propiedades de una clase o estructura son por defecto referencias fuertes, a continuación un ejemplo.

Suponga que se tiene la clase

```
    class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) is being initialized")
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}
```

ahora suponga que se crean las siguientes variables

```
var reference1: Person?
var reference2: Person?
var reference3: Person?
```

si se crea una instancia de person y se realizan las siguientes asignaciones

```
reference1 = Person(name: "John Appleseed")
reference2 = reference1
reference3 = reference1
```

la instancia *John Appleseed* estará fuertemente referenciada por *reference 1, reference2 y reference3* para que el ARC proceda a liberar la memoria que ocupa esta instancia sería necesario que las 3 referencias fueran *nil* de la siguiente manera

```
reference1 = nil
reference2 = nil
reference3 = nil
```

el ARC no desaloja la memoría sino hasta que la última referencia ha sido reasignada a *nil*.

9. Ejemplos

9.1. facil - factorial

```
fun factorial(numero: Int) -> Int{
  if numero == 1{
    return 1
  } else{
    return numero * factorial(numero-1)
  }
}
```

9.2. intermedio - merge sort

```
func elementsInRange<T : Comparable>(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: [])
    }
}
```

9.3. avanzado - segment tree

```
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)
                }
        }
}
```