ETSIT UPM

dit UPM

Desarrollo de Apps para iOS

Introducción a Swift 2

IWEB 2015-2016

Santiago Pavón

ver: 2015.09.20

Lo Básico

Introducción

- Swift es un leenguaje moderno para crear código eficiente, robusto, seguro, …
- Swift es orientado a Objetos
  - Clases: objetos, herencia, polimorfismo, …
  - Y Estructuras, Enumerados, Protocolos, Genéricos, …
- Swift es Type Safe:
  - El tipo de todos los elementos es conocido siempre y se comprueba al compilar.
  - Type inference: Swift infiere el tipo cuando no se indica explícitamente.
- Mayúsculas y minúsculas son distintas.
- Sentencias no acaban en ;
  - ; para separar varias sentencias en la misma línea.

Declarar Variables y Constantes

- **Declarar constantes** con **let**

```
let saludo: String = "hola"
let pi = 3.14
let π = 3.14
let a = 1, msg = "información"
```

- **Declarar variables** can **var**

```
var x = 10
var ❤ = "corazón"
```

  - No hace falta declarar el tipo si puede inferirse del valor asignado.

- ¿Cuándo hay que asignar el valor de una constante?:
  - Se puede declarar una constante sin asignarle un valor, y asignar su valor más tarde, pero siempre antes de que se use la constante.

```
let speed: Double
speed = 25.5
if speed > 10 {
   print("muy rapido")
}
```

- Nota: Antes de Swift 1.2 había que asignar el valor de las constantes cuando se declaran:

```
let speed = 25.5
if speed > 10 {
   print("muy rapido")
}
```

Comentarios

- Hasta el final de línea:

```
let name = "Juan"  // El nombre
```

- Varias líneas:

```
/* una línea
   otra línea
*/
```

  - Los comentarios entre `/*` y `*/` pueden anidarse

Mensajes de Log: **print**

- La función global **print** se usa para sacar mensajes de log por la consola.

```
print(_:separator:terminator)
```

- Toma como parámetros los valores a imprimir:

```
print("hola", 10, x)  // hola 10 algo
```

- Tiene un parámetro opcional para indicar el String a usar como separador de los valores a imprimir (por defecto es " "):

```
print("hola", 10, x, separator: ", ")  // hola, 10, algo
```

- Tiene un parámetro opcional para indicar el String a usar como terminador de líneas (por defecto es "\n"):

```
print("hola", 10, x, terminator: "")
```

Tipos: Enteros

- Tipo Entero es **`Int`**
  - Otros enteros: **`Int8`**, **`Int16`**, **`Int32`**, **`Int64`**, **`UInt`**, **`UInt8`**, **`UInt16`**, **`UInt32`**, **`UInt64`**.
  - Recomendación: usar **`Int`** excepto si se necesita otro por razones importantes.
- Valores máximo y mínimo:

```
Int.max
Int.min
```

Tipos: Punto Flotante

- Tipos:

```
Double      // 64 bits
Float       // 32 bits
```

- Ejemplos:

```
let pi = 3.14 // pi es un Double
```

  - Inferencia de tipos: Double tiene prioridad

```
let perimetro = 2 * pi * radio
```

  - Se infiere Double

Literales

- **Enteros**:
  - Número decimal - sin prefijo
    ```
    let edad = 10
    ```
  - Número binario - con prefijo **0b**
    ```
    let edad = 0b1010
    ```
  - Número octal - con prefijo **0o**
    ```
    let edad = 0o12
    ```
  - Número hexadecimal - con prefijo **0x**
    ```
    let edad = 0xA
    ```
- **Punto flotante**:
  - Número decimal - sin prefijo. La notación científica usa **e** para exponente
    ```
    let x = 1.23e2  // 1.23*10^2, 123
    ```
  - Número hexadecimal - con el prefijo **0x**. La notación científica usa **p** para exponente (potencia de 2)
    ```
    let y = 0xAp3   // 10*2^3, 80
    ```
- Facilitar legibilidad:
  ```
  let sueldo = 2_345_238.345_3  // los _ se ignoran
  ```

Booleanos

- Tipo: **Bool**
- Valores: **true** y **false**

```
let calvo = true
if calvo { // la condición debe ser booleana
   print("no tengo pelo")
} else {
   print("necesito un peine")
}
```

Tipo: **Character**

- Literales entre " y ".

```
let initial: Character = "L"
```

  - Caracteres especiales: **\0**, ****, **\t**, **\n**, **\r**, **\"**, **\'**, **\x*nn***, **\u*nnnn***, **\U*nnnnnnnn*** (códigos unicode, ***n*** es hexadecimal),

- Inicializador:

```
var code = Character("a")
```

Tipo: **String**

- Un String está formado por varios **Character**.
- Literales entre " y ".

```
let name = "Luis"
var address = ""
```

- Inicializador:

```
var address = String()  // String vacío, igual que ""
```

- Los String pueden usar el API de **NSString**.

- Concatenar: + +=

```
var str = "abc"
str += "def"
print(str)   // "abcdef"
```

- Comparar: ==

- Propiedades:
  - **isEmpty**, **utf8**, **utf16**, **unicodeScalars**, **startIndex**, **endIndex**, . . .

- Métodos:
  - **hasPrefix**, **hasSuffix**, **uppercaseString**, **lowercaseString**, **toInt**, **join**, **advance**, **componentsSeparatedByString**, **rangeOfString**, **description**, **splice**, . . .

- Añadir un Character al final de un String:

```
let c: Character = "🚚"
var s = "camión"
s.append(c)    // s es "camión🚚"
```

- Iterar sobre los caracteres (unicode) de un String:

```
for c in s.characters {
    print(c)
}
// c
// a
// m
// i
// ó
// n
// 🚚
```

- Contar los caracteres de un String:
  - La propiedad **count** de la propiedad **characters** del string

```
s.characters.count  // 7
```

- **Interpolación de Strings**:
  - Insertar el valor de una expresión dentro de un String.

```
let name = "Pepe"
let saludo = "Hola \(name), que pases un buen día"
print("Hola \(name) y adios") // Hola Pepe y adios
print(saludo) // Hola Pepe, que pases un buen día
```

  - En el String se sustituye

```
\(expresion)
```

  - por el valor de la expresión.

Conversión de Tipos

- Los valores nunca se convierten implícitamente a otro tipo.

```
let a = 10           // a es un Int
let b: Double = a    // ERROR: No se convierte Int en Double
```

- Hay que crear una instancia nueva del tipo deseado.
  - Para crear la instancia se usa el inicializador del tipo deseado pasando como argumento el valor inicial.

```
let c = Double(a)
let d = Int(pi)   // Los decimales se truncan.
let e = "Valor = " + String(c)
```

Notas

CGFloat

El tipo CGFloat aparecerá muchas veces cuando usemos UIKit.
Para Objective-C es un define a float (de C).
En Swift es una estructura, y disponemos de numerosos inicializadores para realizar el casting a otros tipos: Double, Float, …

```
var a: CGFloat = 2.0

var b: Float = Float(a)

var c: Double = 3.5

a = CGFloat(c)
```

Operadores

- Tenemos los operadores básicos que son típicos en muchos lenguajes:

  `+ - * / % += -= *= /= %= ++ -- ?: == != < > <= >= ! && || !`

  - Pero con algunos cambios.
    - `=` no devuelve un valor.
      - Para evitar confundirlo con `==`.
    - `+`, `-`, `*`, `/`, `%`, etc. detectan y no permiten desbordamiento (overflow).
      - Existen operadores que permiten desbordamiento (`&+`, `&*`, …).
    - `%` calcula el resto de la división, no el módulo.
      - Y puede aplicarse sobre valores reales.
    - Las prioridades no son exactamente iguales a las de otros lenguajes.
    - . . .

- Otros operadores: `&+` `&-` `&*`
  - Versiones de los operadores `+`, `-` y `*` pero permiten **desbordamiento**.
    - Si el valor calculado excede el mayor o menor valor soportado se descartan los bits sobrantes.

```
let a = 5
var b = 2
b = 3*a

let str = "hola" + "y adios"

var x1 = Int.max   //  9223372036854775807
var x2 = x1 &+ 1   // -9223372036854775808
```

Tuplas

- Agrupar varios valores en uno solo.
  - Los valores dentro de la tupla pueden ser de distinto tipo.

```
let resultado = (200, "OK")

// Extraer valores.
let (codigo, texto) = resultado
print("El codigo es \(codigo)")

// Usar _ cuando no interese extraer algún componente.
let (codigo, _) = resultado

// Acceder por índice:
let codigo = resultado.0
let texto = resultado.1

// Crear tupla asignando un nombre a los componentes.
// Accedo usando el nombre.
let resultado = (code: 400, msg: "Not found")
let codigo = resultado.code
let texto = resultado.msg
```

Tipo **Range**

- Para representar un rango de valores.
- Literales y Operadores para indicar **rangos**:

  `a..<b`

  - es el rango de valores desde `a` hasta `b`, pero incluyendo `a` y excluyendo `b`.

  `a...b`

  - es el rango de valores desde `a` hasta `b`, ambos incluidos.

- El tipo Range en realidad es una structura:

```
struct Range<T> {
    var startIndex: Int
    var endIndex: Int
}
```

- Recorridos:

```
for i in 1...4 { print(i) }  // 1 2 3 4
for i in 1..<4 { print(i) }  // 1 2 3
```

Más Tipos

- Optional.
- Arrays, Diccionarios, Sets.
- Clases, Estructuras, Enumerados.
- Closures.

Tipos Valor y Referencia

- **Tipos Valor**:
  - En las asignaciones, paso de parámetros y devolución de tipos se pasa una copia del valor.
    - Las estructuras y los enumerados se manejan por valor.
      - Los **Int**, **Double**, **Bool**, **String**, **Character**, **Range**, **Array**, **Set**, **Dictionary**, … son estructuras.
      - Los **Optional** son enumerados.
- **Tipos Referencia**:
  - En las asignaciones, paso de parámetros y devolución de tipos se pasa la referencia a la instancia.
    - Las **clases** y las **closures** son tipos manejados por referencia.

- Ejemplo:
  - Como el tipo String es una estructura, entonces sus instancias se asignan/pasan/manejan por **valor**, es decir, al asignarlos, pasarlos como parámetro, devolverlos en funciones, etc. se hace una copia del valor.

```
var a = "uno"
var b = a
a = "tres"
print(a)   // "tres"
print(b)   // "uno"
```

Type Aliases

- Crear un nombre alternativo para un tipo ya existente:

```
typealias Edad = Int
```

Assert

- Usado para depurar.
  - En producción los assert se ignoran: es como si no existieran.
- **assert** comprueba que una condición sea true.
  - Si lo es, el programa continúa su ejecución.
  - Si no lo es, se detiene la ejecución del programa.
- Puede pasarse un segundo parámetro con un mensaje explicativo.

```
assert(edad >= 0, "La edad debe ser positiva")
```

Tipo Optional

Tipo **Optional**

- Un tipo **Optional** se usa para indicar que:
  - podemos tener un valor,
  - o podemos no tenerlo.
- En Swift no existe el concepto de puntero que puede apuntar a un dato o a nulo.
  - Se ha sustituido por el uso de tipo optional.
- Para declarar un tipo Optional, añadir **?** detrás del nombre de tipo.

```
let dato: String?
```

- Se usa **nil** para indicar que no hay valor.

```
if dato != nil { . . . }
```

- Cuando hay un valor, se extrae (*unwrapping*) añadiendo **!** detrás del valor Optional.

```
let v: String = dato!
```

```
Usar ! cuando no hay valor, produce un error de ejecución y el
programa se muere.
```

- En realidad, el tipo Optional es un enum genérico con dos casos.

```
enum Optional<T> {
    case None
    case Some(T)
}
```

  - El uso de **?** para declara un Optional es azúcar sintáctica.

- Ejemplo: Rellenamos un formulario con nuestra edad y lo imprimimos por la consola:

```
// str es el string tecleado en el formulario.
// El usuario debería haber metido su edad como un número.
let str = textField.text

// Convertir el String en un Int.
let edad: Int? = str.toInt()

// Si el usuario no tecleo un numero:
if edad == nil {
   print("El formulario se ha rellenado mal")
} else { // El usuario si tecleo un numero:
   print("Tienes \(edad!) años")    // ! extrae el valor
}
```

  - **`toInt`** es un método de los String que devuelve `Int?`.
  - Por la inferencia de tipos, se puede poner: `let edad = str.toInt()`

- El valor por defecto de un Optional es nil.
  - Si no se asigna un valor inicial al declarar una constante o variable Optional, se asigna automáticamente nil.

```
var name: String?
// name se inicializa sin valor, es decir a nil
// Usar name! produciría un error de ejecución

name = "Pepe"
// Ahora name tiene un valor.
// El valor asociado es "Pepe".

print("Me llamo \(name!).")
// Accedo al valor asociado usando !

name = nil
// name vuelve a no tener un valor.
```

Optional Binding

- **Optional Binding**:
  - Sentencia if o while que
  - comprueba si un Optional contiene un valor
  - y si lo contiene, se declara una constante o una variable inicializada con el valor extraído del optional.

```
if let miEdad = str.toInt() {
   print("Tienes \(miEdad) años")
} else {
   print("El formulario se ha rellenado mal")
}
```

    - Si el valor a la izquierda del = es del tipo X, entonces el valor a la derecha del = debe ser un Optional del tipo X.
      - `str.toInt()` es de tipo `Int?`
      - `miEdad` es de tipo `Int`
    - No necesito usar ! para extraer el valor.

- **Optional Binding con varias condiciones**:
  - Desde Swift 1.2 pueden comprobarse varios opcionales de una vez.
  - También se puede usar la cláusula **where** para comprobar condiciones booleanas.

```
var strX = "1"
var strY = "2"
var strZ = "0"

if let x = strX.toInt() where x>0,
   let y = strY.toInt(), z = strZ.toInt() {
       print(x, y, z)
}
```

Implicitly Unwrapped Optional

- Si declaramos una constante o variable Optional pero sabemos que siempre va a tener un valor, podemos declararla usando ! en vez de ?.
- Entonces:
  - Podemos acceder al valor asociado sin usar !

```
let edad: Int! = 33
print(edad)
```

  - Puede compararse con nil (aunque sabemos que no lo es).

```
if edad != nil { … }
```

  - Puede usarse en un Optional Binding.

```
if let años = edad { … }
```

Optional Chaining

- Es una forma consultar o llamar a propiedades, métodos o subscripts con valores Optional.
  - Si el Optional no es nil, se usa la propiedad, método o subscript normalmente.
  - Si el Optional es nil, la llamada a la propiedad, método o subscript devuelve nil.
- Este tipo de consultas o llamadas pueden encadenarse.
  - Si algún elemento de la cadena devuelve nil, la cadena se interrumpe.
- Se añade ? detrás de los elementos que pueden ser nil.

```
var name = myself.car?.garage()?.owner.father?.sons?[0].name
// Nombre del hermano mayor del dueño del taller donde llevo mi coche.
// Si algún elemento de esa cadena es nil, entonces name será nil.
```

- El tipo devuelto por un Optional Chaining siempre es Optional,
  - ya que existe la posibilidad de que devuelva un valor o nil.
  - El tipo devuelto por un Optional Chaining es el tipo normal devuelto por la propiedad, método o subscript llamado, pero encapsulado en un Optional.
    - Si el tipo devuelto ya era un Optional, no se hace nada: Ya tenemos un Optional.

- Optional chaining puede usarse también en la asignación de un valor a una propiedad directamente, o usando subscripts:

```
myself.car?.year = 2014
```
- Esta asignación no se realiza si yo no tengo coche.

```
garage.cars?[0].year = 2014
```
- Esta asignación no se realiza si no hay coches en el garage (*garage.cars es nil*)

- Si un método o un subscript devuelve un Optional, la interrogación va detrás de los corchetes o paréntesis.

```
store["ajax"]?.price = 100
```
- La búsqueda de "ajax" en la tienda puede devolver nil.

```
store.find("ajax")?.price = 100
```
- La búsqueda de "ajax" en la tienda puede devolver nil.

Operador: Nil Coalescing

- `a ?? b`
  - Si el valor **optional** `a` tiene un valor, entonces devuelve el valor asociado de `a`, en caso contrario devuelve `b`.
  - Es equivalente a escribir: `a != nil ? a! : b`

- Ejemplo:

```
let m:String?
let n = m ?? "defecto"  // n es "defecto"
```

Arrays

Arrays

- Un array es una lista ordenada de valores del mismo tipo.
- Puede crearse un array para almacenar cualquier tipo de valor.
  - Todos los valores del array creado son del mismo tipo.
- Recordar que Swift es **type safe**, luego:
  - El tipo de los valores de un array siempre es conocido.
    - No puede insertarse un valor de otro tipo en el array.
- Se accede a los elementos de un array por su índice.
  - Acceder a un índice inexistente provoca un error de ejecución y la muerte del programa.
- Un **Array** declarado con **let** es constante, y declarado con **var** es mutable.

- Literales:

```
[]                    // Literal array vacío.
["Pepe", "Luis"]      // Literal con varios valores.
```

- Crear un array:

```
var names1: [String] = ["Pepe", "Luis"]

var names1: Array<String> = ["Pepe", "Luis"]

var names1 = ["Pepe", "Luis"]  // Se infiere el tipo.

var names2 = [String]()    // Inicialización. Vacío.

var names3 = [String](count:5, repeatedValue:"John")
                        // Crear repitiendo un valor.

var names4 = names1 + names1    // Concatenar

var numbers = Array(1...5)      // Crea [1,2,3,4,5]
```

- Para acceder y modificar el array usar métodos, propiedades o la sintaxis subscript:

```
names.count   // propiedad read-only. Tamaño del array.
names.isEmpty // propiedad que indica si tamaño es 0.
names.append("Carlos")  // método para añadir al final.
names += ["Pedro", "Ana"] // añadir nombres al final.
names[1] = "Ines"   // cambiar segundo nombre.
let n = names[3]    // acceder al cuarto nombre.
names[2...4] = ["Eva", "Lupe"]  // sustituir 3 nombres
                                // por 2 nombres.
names.insert("Felipe", atIndex:3)
            // insertar en una posición.
names.removeAtIndex(2)
            // eliminar un nombre. No se dejan huecos.
names.removeLast()  // eliminar el último nombre.

// first, last, splice, sort, sorted, map, flatMap,
// reduce, filter, …
```

- Acceder a un elemento inexistente de un array produce un error de ejecución.

```
var z = [1,2]
z[3]   // ERROR DE EJECUCIÓN
```

- Recorrer un array:

```
for name in names {
    print(name)
}

for (index, value) in enumerate(names) {
    print("Posición \(index): \(value)")
}
   // enumerate es una función global.
```

Sets

Sets

- Un **Set** es un **conjunto** de valores del mismo tipo.
  - En un conjunto los valores no están duplicados, ni están ordenados.
- Puede almacenar cualquier tipo de valor.
  - Pero todos los valores de un Set son del mismo tipo.
  - Recordar que Swift es **type safe**, luego:
    - El tipo de los valores de un Set siempre es conocido.
      - No puede añadirse un valor de otro tipo en el Set.
- Con los Sets se pueden realizar las operaciones típicas de conjuntos:
  - Comprobar si contienen un valor, enumerar todos los valores, hacer la unión con otro Set, …
- Un **Set** declarado con **let** es constante, y declarado con **var** es mutable.

- Literales: *(se crean con un literal de arrays)*

```
let s: Set<String> = []   // Hay que declarar el tipo.
let s: Set = ["Pepe", "Luis"]  // Conjunto de Strings.
```

- Crear un Set:

```
var s1: Set<String>()
let s2: Set = ["Pepe", "Luis"]
var s3 = Set(["Pepe", "Luis"])
```

- Añadir y eliminar elementos:

```
s3.insert("Jaime")  // s3 es ["Pepe", "Luis", "Jaime"].
s3.remove("Pepe")   // s3 es ["Luis", "Jaime"].
          // remove devuelve un Optional con
          // el valor eliminado o nil.
s3.removeAll()      // s3 es [].
```

- Tamaño:

```
s3.count   // Propiedad read-only.
           // Número de elementos contenidos en el Set.
s3.isEmpty // propiedad que indica si tamaño es 0.
```

- Pertenece, contiene, disjuntos, …:

```
var s0 = Set(["Uno", "Dos", "Tres"])
var s1 = Set(["Uno", "Dos", "Tres"])
var s2 = Set(["Uno", "Dos"])
var s3 = Set(["Uno", "Dos", "Cuatro"])
var s4 = Set(["Cinco", "Seis"])

s1.contains("Dos")          // true  (contiene elemento)

s1.isSubsetOf(s0)           // true  (subconjunto)
s2.isSubsetOf(s0)           // true
s1.isStrictSubsetOf(s0)     // false (subconjunto pero no igual)
s2.isStrictSubsetOf(s0)     // true

s0.isSupersetOf(s2)         // true (superconjunto)
s0.isStrictSupersetOf(s2)   // true (superconjunto pero no igual)
s0.isStrictSupersetOf(s3)   // false

s0.isDisjointWith(s2)       // false (disjuntos)
s0.isDisjointWith(s4)       // true
```

NOTA: Estos métodos también aceptan un array como argumento:

```
var a = ["Cinco", "Seis"]
s2.isSubsetOf(a)              // false
```

- Unión, intersección, ...:

```
s0.union(s3) // Devuelve ["Uno", "Dos", "Tres", "Cuatro"]
s0.subtract(s3) // Devuelve ["Tres"]
s0.intersect(s3) // Devuelve ["Uno", "Dos"]
s0.exclusiveOr(s3) // Devuelve ["Tres", "Cuatro"]
```

- InPlace:

  Existe una versión de estos métodos con el mismo nombre acabado en `InPlace` que modifican el valor de la variable sobre la que se aplican:

```
s0.unionInPlace(s3)      // Modifican s0.
s0.subtractInPlace(s3)   // s0 debe declararse con var.
s0.intersectInPlace(s3)
s0.exclusiveOrInPlace(s3)
```

- Recorrer un Set:

```
for n in s0 {
    print(n)
}
```

Diccionarios

Diccionarios

- Un diccionario: es una colección que
  - Almacena valores asociados a una clave.
  - Todas las claves son de un mismo tipo.
  - Todos los valores son de un mismo tipo.
- Recordar que Swift es **type safe**, luego:
  - El tipo de las claves y valores de un diccionario siempre es conocido.
    - No puede insertarse una pareja clave-valor de otros tipos.
- Se accede a los valores del diccionario por su clave.
  - El tipo devuelto es un Optional dado que la clave buscada puede no existir.
- Un diccionario declarado con **let** es constante, y declarado con **var** es mutable.

- Literales:

```
[:]                                  // diccionario vacío
["M":"Madrid", "B":"Barcelona"]      // parejas clave:valor
```

- Crear un diccionario:

```
var codes: [String:String] = ["M":"Madrid",
                              "B":"Barcelona",
                              "CR":"Ciudad Real"]

var codes: Dictionary<String,String> = ["M":"Madrid",
                                        "B":"Barcelona",
                                        "CR":"Ciudad Real"]

var codes = ["M": "Madrid",    // se infiere el tipo
             "B": "Barcelona",
             "CR": "Ciudad Real"]

var codes = [String: String]()  // Inicialización. Vacío.
```

- Acceder y modificar el diccionario usando métodos, propiedades o la sintaxis subscript:

```
codes.count                  // propiedad. Tamaño.

codes.isEmpty                // propiedad. ¿Tamaño es 0?

codes["V"] = "Valencia"  // crear o actualizar dato.

codes.updateValue("Jaen", forKey:"J")
        // devuelve un String? con el valor antiguo o nil.

let n = codes["J"]
        // devuelve un String? con el valor o nil.

codes["J"] = nil  // eliminar clave y valor.

let v = codes.removeValueForKey("J")
        // devuelve un String? con el valor eliminado o nil.
```

- Recorrer un diccionario:

```
for (k,v) in codes {
   print("clave \(k) - valor \(v)")
}

for k in codes.keys {
   print("clave \(k)")
}
   // keys: propiedad con un iterable de las claves.
   // Puede crearse un array con [String](codes.keys)

for v in codes.values {
   print("valor \(v)")
}
   // values : propiedad con un iterable de los valores
   // Puede crearse un array con [String](codes.values)
```

- El tipo de la clave debe ser conforme con el protocolo **Hashable**.
  - Proporciona:
    - propiedad **hashValue: Int**
    - operador **==**
  - Los tipos básicos de Swift (String, Int, Double, Bool) son hashable.
    - También los miembros de las enumeraciones sin valores asociados.

Framework Foundation

Framework Foundation

- Proporciona:
  - Clases
  - Notificaciones
  - Sistema de ficheros
  - Threads
  - Comunicaciones
  - etc.
- Consultar la documentación/help para ver los detalles.

Algunas Clases de Foundation

- **`NSObject`**
  - Es la clase base de todas las clase Objective-C.
  - En ocasiones puede ser necesario que nuestras clases Swift deriven de esta clase para incorporar algunas características avanzadas.
- **`NSNumber`**
  - Clase envoltorio para los tipos numéricos (Int, Float, Double, Bool, …).

```
let a = NSNumber(0.5)
let b = n.floatValue
```

- **`NSValue`**
  - Envoltorio de cualquier tipo de dato.
- **`NSData, NSMutableData`**
  - Buffer de bytes.

- **NSString, NSMutableString**
  - Compatibles con el tipo **String**.

```
let str: String? = NSString(data: data!,
                            encoding: NSUTF8StringEncoding)
```

- **NSArray, NSMutableArray**
  - Compatibles con el tipo **Array<AnyObject>**.

```
var arr = NSArray(contentsOfFile:path) as [Int]
```

- **NSDictionary, NSMutableDictionary**
  - Compatibles con el tipo **Dictionary<NSObject,AnyObject>**.

```
(dic as NSDictionary).writeToFile(path, atomically:true)
```

- **NS[Mutable][Ordered]Set, NSCountedSet**
  - Conjuntos.
- **NSHashTable, NSMapTable**

- **`NSDate, NSCalendar, NSDateFormatter, NSDateComponents`**
  - Fechas y horas.
- **`NSNumberFormatter`**
  - Formatear números.
- **`NSRegularExpression`**
  - Expresiones regulares.
- **`NSOperation, NSOperationQueue, ...`**
  - Operaciones concurrentes.
- **`NSCache`**
  - Colección de pares clave-valor similar a un NSDictionary.
  - Puede liberarse la memoria de los objetos que contiene si hay escasez de memoria.
- **`NSUserDefaults`**
  - Preferencias de usuario.
- **`NSProgress`**
  - Informes de progreso.
- . . .

Control de Flujo

Bucle: **for-in**

- Iterar sobre colecciones, rangos de números, caracteres de un String:

```
for n in 1...10 {
   print(n)
}
   // n se declara automáticamente como una constante.
   // No es necesario poner let.

for c in "Hola" {
   print(c)
}

for v in array {
   print(v)
}

for (k,v) in diccionario {
   print("clave \(k) - valor \(v)")
}
```

Rangos

- Rango incluyendo valores izquierdo y derecho

  1...10

- Rango incluyendo el valor izquierdo y excluyendo el derecho

  1..<10

```
for n in 1...10 {
   print(n)      // 1 2 3 4 5 6 7 8 9 10
}

for n in 1..<10 {
   print(n)      // 1 2 3 4 5 6 7 8 9
}
```

Bucle: inicializa-condición-incremento

```
for var i = 0 ; i <10 ; ++i {
    print(i)
}
```

Bucle: **while**

- Iterar hasta que la condición es false.

```
var n = 10

while n > 0 {
    n--
}
```

Bucle: **repeat-while**

- Iterar mientras que la condición es true.

```
var n = 10

repeat {
    n--
} while n > 0
```

Condicionales

- Ejecuta unas sentencias si la condición es true.
  - La condición debe ser una expresión booleana.
- Puede llevar un else para el caso de false

```
if n > 0 {
    print("positivo")
} else {
    print("no es positivo")
}
```

switch

- Compara un valor con varios patrones para decidir que debe ejecutar.

```
let n = 7
switch n {
case 0:
   print("cero")
case 1, 2, 3:
   print("pequeño")
case 4:
   print("grande")
default:    // el switch debe cubrir todos los valores posibles
   print("otro")
}
```

- Cada punto de entrada debe tener por lo menos una sentencia.
- La ejecución de un case no continua en el siguiente case (o default).
  - Usar **fallthrough** pasa continuar en el siguiente case (o default).
- Si la expresión del switch encaja con varios cases, entonces se entra solo en el primero.

- En el case pueden usarse rangos:

```
let age = 22

switch age {
case 0:
  print("recien nacido")
case 1...12:
  print("Niño")
case 13...18:
  print("Adolescente")
default:
  print("Viejo")
}
```

- Usar tuplas para comprobar varios valores.

```
let p = (2, 4)

switch p {
case (0,0):
   print("En el origen")
case (_,0):
   print("En el eje X")
case (0,_):
   print("En el eje Y")
case (-1...1, -1...1):
   print("Cerca del origen")
default:
   print("En otro sitio")
}
```

  - El _ encaja con cualquier valor.

- Value bindings: Enlazar los valores que comprueba el switch con variables y constantes.

```
let p = (2, 4)

switch p {
case (let x, 0):
   print("En el eje X con x igual a \(x)")
case (0, let y):
   print("En el eje Y con y igual a  \(y)")
case let (x, y):
   print("En el punto x = \(x) e y = \(y)")
}
```

  - No hace falta default porque el último case cubre todos los casos.
  - En el ejemplo declaro constantes, pero podría declarar variables usando var.

- Usar la cláusula **where** para comprobar condiciones adicionales:

```
let p = (2, 4)

switch p {
case let (x, y) where x == y:
   print("En la primera diagonal")
case let (x, y) where x == -y:
   print("En la segunda diagonal")
case let (x, y):
   print("En otro sitio")
}
```

Transferir el Control

- **`continue`**
  - Terminar la ejecución de la iteración actual de un bucle y comenzar con la siguiente iteración.
- **`break`**
  - Terminar la ejecución de un bucle o un switch, y continuar con la siguiente instrucción.
- **`fallthrough`**
  - En un case, indica que se debe continuar ejecutando las sentencias del siguiente case (o default).

- **Etiquetas**:
  - Los bucles y los switch pueden etiquetarse con un nombre.
  - Las sentencias continue y break pueden usar esa etiqueta para indicar a que bucle o switch se refieren.

```
let tablero = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9] ]

fuera: for var x = 0 ; x < 3 ; x++ {
    for var y = 0 ; y < 3 ; y++ {
        if tablero[x][y] == 5 {
            print("Encontrado")
            break fuera
        } else {
            print("Sigo buscando")
        }
    }
}
print("Estoy fuera")
```

Sentencia **guard**

- Es una sentencia que suele usarse para abandonar el ámbito de ejecución actual cuando no se cumple una condición.
  - El código es más sencillo que usar múltiples if-then anidados.
- La sentencia **guard** tiene una **condición booleana** y un **else**.
  - Si la condición del guard es **true**, se ejecutan las sentencias que siguen a la sentencia guard.
    - Cualquier variable o constante asignada en la condición del guard usando optional binding, está disponible en las sentencias después del guard.
  - La cláusula else se ejecuta si la condición es **false**.
    - Las sentencias de la parte else deben provocar que se abandone el ámbito de la sentencia guard, usando return, break, continue, throw, fatalError(), …

- Ejemplo usando if-else:

```
func demo0(dic: [String:String]) {
    if let name = dic["name"] {
        if let address = dic["address"] {

            print("La dirección de \(name) es \(address)")

        } else {
            print("No hay dirección")
         }
    } else {
        print("No hay nombre")
    }
}


demo0(["name":"Pepe", "address":"Plaza España"])
                // La dirección de Pepe es Plaza España
```

- Con sentencias **guard** queda más claro el código:

```
func demo1(dic: [String:String]) {
    guard let name = dic["name"] else {
        print("No hay nombre")
        return
    }
    guard let address = dic["address"] else {
        print("No hay dirección")
        return
    }
    print("La dirección de \(name) es \(address)")
}

demo1(["name":"Pepe", "address":"Plaza España"])
            // La dirección de Pepe es Plaza España
```

- Más compacto y claro:

```
func demo2(dic: [String:String]) {
    guard let name = dic["name"],
          let address = dic["address"] else {
        print("No hay nombre o dirección")
        return
    }
    print("La dirección de \(name) es \(address)")
}

demo2(["name":"Pepe", "address":"Plaza España"])
                // La dirección de Pepe es Plaza España
```

Sentencia **defer**

- **defer** se usa para proporcionar un conjunto de sentencias que deben ejecutarse al abandonar el ámbito de ejecución actual.

```
defer {
    sentencias
}
```

  - Las sentencias del defer no pueden transferir el control fuera de ellas: no pueden llamar a break, return , etc…

- Las sentencias del **defer** se ejecutan siempre, independientemente de como se salga del ámbito de ejecución.
  - Aunque se salga del ámbito por return, break, lanzando un error, … siempre se ejecutan las sentencias del **defer** programado.

- Muy útil cuando se quieren liberar recursos adquiridos previamente al comienzo del ámbito, o finalizar tareas, cerrar descriptores, …
  - Permite escribir programas más claros de entender.

```
func sinDefer() {
    bloquearRecurso()
    if condicion1 {
        liberarRecurso()
        return
    }
    do {
        try tarea()
    } catch {
        liberarRecurso()
        return
    }
    if condicion2 {
        liberarRecurso()
        return
    }
    liberarRecurso()
}
```

```
func conDefer() {
    bloquearRecurso()
    defer {
        liberarRecurso()
    }
    if condicion1 {
        return
    }
    do {
        try tarea()
    } catch {
        return
    }
    if condicion2 {
        return
    }
}
```

Disponibilidad de las APIs

- El compilador comprueba que todas las APIs usadas están disponibles en los dispositivos donde se puede instalar la aplicación.
  - Estos dispositivos son los **Deployment Target** especificados en el proyecto.
- El uso de un API que no esté disponible produce un error de compilación.
- Usaremos una condición de disponibilidad en un **if**, **while** o **guard**, para controlar que código se ejecutaremos dependiendo de la plataforma donde despleguemos:

```
if #available(iOS 9, OSX 10.10, *) {
    // Sentencias a ejecutar si las APIs están disponibles.
} else {
    // Alternativa si no están disponibles.
}
```

  - **#available** toma una lista de plataformas (**iOS**, **OSX** o **watchOS**) y versiones mínimas, terminando con * para indicar otras plafaformas).

Funciones

Funciones

- Es un fragmento de código autocontenido que realiza una cierta tarea.
- Cada función:
  - Tiene un nombre que la identifica.
  - Puede tener cero o más parámetros.
    - cada parámetro puede tener nombres.
    - cada parámetro tiene un tipo.
  - Puede tener un tipo de retorno.
- Cada función tiene un tipo, formado por el tipo de los parámetros y el tipo de retorno.
  - Los tipos función pueden usarse en los mismos sitios que cualquier otro tipo.
- Las funciones pueden anidarse:
  - Una función anidada se define dentro del cuerpo de otra función.
  - Por defecto, solo se ven dentro de la función donde se han definido.

- Definir una función:

```
func wellcome(name: String) -> String {
    let msg = "Bienvenido \(name)."
    return msg
}
```

  - Si no hay parámetros de entrada poner solo los paréntesis **()**.
  - Si no se devuelve nada, omitir la flecha y el tipo de retorno.
    - En realidad se devuelve una tupla vacía del tipo **Void**.

- Invocar la función:

```
let m = wellcome("Pedro")
```

- Si una función devuelve varios valores, usar tuplas.

```
func find(city: String)
            -> (latitude: Double, longitude: Double) {
   // buscar la ciudad
   let lat = 40.1234
   let lon = 3.7123
   return (lat, lon)
}
```

  - En la definición de la función se definen nombres para los componentes de la tupla a devolver.
  - En la sentencia return no es obligatorio usar los nombres de los componentes de la tupla devuelta.

```
let pos = find("Madrid")
print("Posicion= \(pos.latitude) , \(pos.longitude)")
```

- La función anterior estaría mejor si devolviera una tupla opcional dado que puede ocurrir que no se encuentre la ciudad:

```
func find(city: String)
                -> (latitude: Double, longitude: Double)? {
   if tengo_suerte {
      let lat = 40.1234
      let lon = 3.7123
      return (lat, lon)
   } else {
      return nil
   }
}

if let pos = find("Madrid") {
   print("Posición= \(pos.latitude) , \(pos.longitude)")
} else {
   print("No encontrada")
}
```

Nombres de los Parámetros

- Los parámetros de las funciones tienes dos nombres:
  - Un nombre **externo** que se usa en las llamadas a la función.
  - Un nombre **interno** que se usa en el cuerpo de la función, y que debe ser único
- Pero por defecto:
  - No se usa el nombre externo del primer parámetro en las llamadas a la función.
  - Para el segundo y siguientes parámetros, se usa el nombre interno como nombre externo.
- Ejemplo:

```
func unir(p1: String, p2: String) -> String {
   return "\(p1) - \(p2)."
}
unir("uno", p2:"dos")  // uno - dos
```

  - p1 es el nombre interno y externo del primer parámetro.
    - El nombre externo del primer parámetro no se usa en la llamada a la función.
  - p2 es el nombre interno y externo del primer parámetro.

- Podemos proporcionar un nombre de parámetro externo escribiéndolo delante del parámetro interno:

```
func unir(p1: String, con p2: String) -> String {
    return "\(p1) - \(p2)."
}
unir("uno", con:"dos")  // uno - dos
```

- Si es especifica el nombre externo de un parámetro, entonces debe usarse siempre en las llamadas a la función,
  - incluso si es el primer parámetro.

- Si no se quiere usar el nombre externo de un parámetro (el segundo y siguientes), usar "_" como nombre externo en la declaración del parámetro.

```
func unir(p1: String, _ p2: String) -> String {
    return "\(p1) - \(p2)."
}
unir("uno","dos")  // uno - dos
```

Valor por defecto de parámetro

- Al definir una función, puede definirse un valor por defecto para los parámetros.
  - Solo para los parámetros situados al final de la lista de parámetros.

```
func wellcome(name: String,
              withMessage msg: String = "Bienvenido") -> String {
   let msg = "\(msg) \(name)."
   return msg
}

wellcome("Pedro", withMessage: "Hi")  // Hi Pedro
wellcome("Pedro")                     // Bienvenido Pedro
```

- Si no se define un nombre externo para los parámetros con valores opcionales, Swift lo define automáticamente y tenemos que usarlo.
  - Usa como nombre externo el nombre local, como si hubiéramos escrito un **#**.
  - Este comportamiento puede anularse proporcionado un **_** (subrayado) como nombre externo del parámetro.

Número Variable de Parámetros

- *Variadic Parameter*: es un parámetro que acepta cero o más valores de un determinado tipo.
  - Añadiendo tres puntos después del tipo.
  - Los valores pasados en este parámetro son accesibles como un **array** del tipo indicado.

```
func sum(numbers: Int...) -> Int {
    var total = 0
    for n in numbers {
        total += n
    }
    return total
}

let money = sum(22, 34, 11)  //  67
```

  - Una función solo puede tener como máximo un parámetro variadic.

Parámetros: Constantes o Variables

- Por defecto, los parámetros de una función son constantes, es decir, no puede cambiarse su valor en el cuerpo de la función.

```
func next(value: Int) -> Int { // calcula el siguiente entero
    value += 1   // ERROR: value es una constante
    return value
}
```

- Para definir un parámetro variable anteponer **var**.
  - Los cambios realizados en un parámetro variable no se ven fuera del cuerpo de la función.

```
func next(var value: Int) -> Int {
    value += 1  // OK: value es una variable
    return value
}
next(8)  // devuelve 9
```

Parámetros In-Out

- Para que los cambios realizados en un parámetro persistan tras finalizar la ejecución de una función:
  - Definir el parámetro como **inout**.
- En la llamada a la función, solo pueden pasarse variables en los parámetros inout.
  - añadiendo un **&** delante del nombre de la variable.

```
func next(inout value: Int) {
    value += 1
}

var x = 8
next(&x)
print(x)  // 9
```

```
func swap(inout a: Int, inout b: Int) {
    (b,a) = (a,b)
}

func increment(inout value: Int, by: Int = 1) {
    value += by
}

var a1 = 2
var a2 = 4

swap(&a1, &a2)

a1  // 4
a2  // 2

increment(&a1, by:3)

a1  // 7

increment(&a1)

a1  // 8
```

Tipo Función

- Cada función tiene un tipo.
  - Formado por el tipo de los parámetros y el tipo de retorno.
- El tipo de

```
func sum(a: Int, b: Int) -> Int { return a+b }
```

  - es **`(Int, Int) -> Int`**
    - Este tipo significa: función que toma dos enteros y devuelve un entero.
- El tipo de

```
func hello() { print("Hi") }
```

  - es **`() -> ()`**
    - Este tipo significa: función que no toma parámetros y devuelve **Void**.

- Un tipo función puede usarse igual que cualquier otro tipo:
  - puede ser el tipo de una variable o constante, el tipo del parámetro de una función, el tipo del valor devuelto por una función, etc...

```
func sum(a: Int, b: Int) -> Int {
   return a+b
}

var ope = sum  // ope es de tipo (Int, Int) -> Int

ope(2,9)       // Invoco la función guarda en ope

func apply(v1: Int,
           v2: Int,
           operation: (Int, Int) -> Int) -> Int {
    return operation(v1, v2)
}  // el tercer parámetro es de tipo (Int, Int) -> Int

apply(10, 20, sum) // Paso un función en último parámetro
```

```swift
let table = [2, 44, -3, 0 , 9, -2]

func positive(v: Int) -> Bool {           // (Int) -> Bool
    return v > 0
}

// Imprime solo algunos valores del array.
// Los que cumplen la condición pasada como parámetro.
func filter(values: [Int], condition: (Int) -> Bool) {

    for v in values {
        if condition(v) {
            print(v)
       }
    }
}

filter(table, positive)  // Imprime 2, 44 y 9
```

Closures

Closures

- Son un tipo de dato cuyo valor son sentencias de código.
- Estos tipos de datos pueden usarse como otro cualquier dato:
  - en variables, en parámetros, etc.
- Estos datos pueden ejecutarse
  - Se ejecutan en el mismo ámbito en el que fueron creados.
    - Al crearlos capturan las variables, constantes, etc. que existen en ese ámbito.
    - y pueden acceder a esas variables, constantes, etc. cuando se ejecutan.
- Los closures pueden ser:
  - **funciones globales**
    - es una closure con un nombre, y no captura ningún valor.
  - **funciones anidadas**
    - es una closure con un nombre, y captura los valores de la función donde se ha definido.
  - **expresiones closure**
    - es una expresión formada por sentencias, que no tiene un nombre, y que captura los valores del contexto donde se creó.
- Las closures (y las funciones) se manejan por referencia.
  - Los valores se pasan por referencia.

Expresión Closure

- La sintaxis de una expresión closure es:

```
{ ( Parámetros ) -> TipoDeRetorno in
  Sentencias
}
```

  - Notad que todo está contenido entre llaves.
  - Los parámetros pueden ser constantes, variables, in-out, tuplas, variadic, o puede no haber parámetros.
    - No se permiten parámetros con valores por defecto.
  - La palabra **in** está situada antes de las sentencias.

```
{ (v: Int) -> Bool in
    return v > 0
}
```

Inferir Tipo

- Las expresiones closure se pueden simplificar si por inferencia de tipos sabemos cuál es el tipo de los parámetros y el tipo de retorno.

```
let table = [2, 44, -3, 0 , 9, -2]

// Ordena un array por el método de la burbuja.
// El criterio de ordenación se pasa como parámetro.
func sort(var values: [Int], condition: (Int,Int) -> Bool) -> [Int] {
    let n = values.count-1
    for var i = 0 ; i<n ; i++ {
        for var j = 0; j < n; j++ {
            if condition(values[j], values[j+1]) {
                (values[j], values[j+1]) = (values[j+1], values[j])
            }
        }
    }
    return values
}
let t1 = sort(table, {(v1: Int, v2: Int) -> Bool in return v1 > v2})
let t2 = sort(table, {v1, v2 in return v1 > v2})  // Más sencillo
```

- He eliminado los paréntesis, el tipo de los parámetros, la flecha y el tipo de retorno, ya que por inferencia de tipos conozco el tipo.

Omitir return

- Si una expresión closure:
  - solo contiene una sentencia **return valor**.
  - y no hay ambigüedad sobre cuál debe ser el tipo del valor devuelto.
  - Entonces puede omitirse la palabra **return**.

```
let t3 = sort(table, {v1, v2 in v1 > v2})
```

Omitir Nombres de los Parámetros

- Si en una expresión closure se puede inferir su tipo,
  - entonces también puede omitirse la lista de parámetros, y usar los nombres **`$0`**, **`$1`**, **`$2`**, etc. para referirse al primer argumento, al segundo, al tercero, etc…
  - también se omite la palabra **in**.

```
let t4 = sort(table, {$0 > $1})
```

Usar un Operador

- Dado que los operadores son funciones.
  - Si un operador encaja con el tipo esperado en un parámetro de una función, entonces puede utilizarse directamente el nombre del operador `>`.

```
let t5 = sort(table, >)
```

    - Aclaración: En realidad, en este ejemplo se está usando una función en un sitio donde se esperaba una función de ese mismo tipo. No es nada especial.

Trailing Closures

- Si una función toma como último parámetro una closure, se puede invocar la función poniendo la expresión closure fuera de los paréntesis.

```
sort(table) {v1, v2 in v1 > v2}
```

  - Nota: Si el único parámetro de la función es la closure, pueden omitirse los paréntesis en la llamada a la función. Solo se pone la expresión closure después del nombre de la función, sin los paréntesis.

Capturar Valores

- Un closure puede capturar las variables y constantes del ámbito donde se definió.
  - La closure puede usar y modificar los valores capturados aunque el ámbito donde se creó ya no exista.

```
func makeAcumulator() -> (Int) -> () {

    var total = 0  // variable local de la función

    return { (v:Int) in
        total += v                    // El closure captura total
        print("Acumulador = \(total).")
    }
}

let ac = makeAcumulator()   // ac es del tipo (Int) -> ()

ac(3) // total es 3
ac(6) // total es 9
ac(8) // total es 17
```

Cuidado con no crear bucles de retenciones

Clases

Clases

- Las clases:
  - Son tipos manejados por referencia.
    - Las instancias se asignan o pasan por referencia.
    - Se usa un contador de referencias en las instancias.
  - Tienen propiedades, métodos, inicializadores, deinicializadores.
  - Soportan herencia.
    - Upcasting, downcasting, polimorfismo, . . .
  - Más cosas:
    - Pueden extenderse.
    - Pueden ser conformes a protocolos.
    - Pueden usar la sintaxis subscript.
    - …

Definir Clases

- Definir una clase:

```
class Figure {
    var name: String?
    var closed = false
    var sides = 0
}
```

- Esta clase define tres propiedades almacenadas (*stored*) variables (*var*), y sus valores iniciales.
  - El valor inicial de name es nil al ser un optional.
  - Los tipos de closed y sides se infiere por los valores iniciales.

Crear Instancias

- Crear instancias usando los inicializadores:

```
let f = Figure()
```

- Este ejemplo usa el inicializador sin parámetros.
  - Las propiedades se inicializan con el valor por defecto.

Acceder a las propiedades

- Usar notación punto para acceder a las propiedades de las instancias:

```
f.name = "triángulo"
f.sides = 3
print("La figura tiene \(f.sides) lados.")
```

Tipos Valor y Referencia

- Tipos Valor: las estructuras y los enumerados.
  - En las asignaciones, paso de parámetros y devolución de tipos se pasa una copia del valor.
- **Tipos Referencia: Las clases.**
  - En las asignaciones, paso de parámetros y devolución de tipos se pasa la referencia a la instancia.

```
let f1 = Figure()
f1.name = "Triángulo"

let f2 = f1

print("\(f1.name!)")  // Triángulo
print("\(f2.name!)")  // Triángulo

f1.name = "Cuadrado"

print("\(f1.name!)")  // Cuadrado
print("\(f2.name!)")  // Cuadrado

// f1 y f2 apuntan/referencian al mismo objeto
```

Comparar Clases

- Para ver si dos instancias de una clase son la misma instancia o son dos instancias equivalentes se usan operadores diferentes.
- Operadores para comparar si dos variables (o constantes) apuntan a la misma instancia:

  ===
  !==

- Operadores para comparar si dos instancias son equivalentes o iguales en función del valor de sus propiedades:

  ==
  !=

  - Puede ser necesario definir estos operadores, o derivar de NSObject, o . . .

Estructuras

Estructuras

- Son tipos manejados por valor.
  - Las instancias se asignan o pasan por valor.
- Tienen propiedades, métodos e inicializadores.
  - No tienen métodos Deinicialzadores.

- No soportan herencia.
  - No aplican los concepto de upcasting, downcasting, polimorfismo, . . .

- Puede usar la sintaxis subscript.
- Pueden extenderse.
- Pueden ser conformes a protocolos.

Definir Estructuras

- Definir un tipo estructura:

```
struct Color {
    var red = 0
    var green = 0
    var blue = 0
}
```

- Este ejemplo define una estructura con tres propiedades almacenadas (*stored properties*) enteras (*inferencia de tipos*) inicializadas a cero.

- En este ejemplo las propiedades son variables (*var*), pero también podrían ser constantes (*let*).

Crear Instancias

- Crear instancias usando los inicializadores:

```
var c = Color()
```

- Este ejemplo usa el inicializador sin parámetros.
  - Las propiedades se inicializan con el valor por defecto.
- En las estructuras se crea automáticamente un inicializador (***memberwise initializator***) que permite dar una valor inicial a las propiedades de la instancia creada:

```
let c2 = Color(red: 50, green: 255, blue: 0)
```

Acceder a las propiedades

- Usar notación punto para acceder a las propiedades de las instancias:

```
c.blue = 100
print("Color = \(c.red) \(c.green) \(c.blue)")
```

Tipos Valor y Referencia

- **Tipos Valor: las estructuras y los enumerados.**
  - En las asignaciones, paso de parámetros y devolución de tipos se pasa una copia del valor.
- Tipos Referencia: Las clases.
  - En las asignaciones, paso de parámetros y devolución de tipos se pasa la referencia a la instancia.

```
var c1 = Color(red: 50, green: 50, blue: 50)
var c2 = c1

print("\(c1.red) \(c1.green) \(c1.blue)")   // 50 50 50
print("\(c2.red) \(c2.green) \(c2.blue)")   // 50 50 50

c1.blue = 100

print("\(c1.red) \(c1.green) \(c1.blue)")   // 50 50 100
print("\(c2.red) \(c2.green) \(c2.blue)")   // 50 50 50

                    // La componente azul de c2 no ha variado
```

```
var c = Color(red: 50, green: 50, blue: 50)
let u = Umbrella() // Suponemos que el paraguas tiene la propiedad color

u.color = c

print("\(c.red) \(c.green) \(c.blue)")                    // 50 50 50
print("\(u.color.red) \(u.color.green) \(u.color.blue)") // 50 50 50

c.blue = 100

print("\(c.red) \(c.green) \(c.blue)")                    // 50 50 100
print("\(u.color.red) \(u.color.green) \(u.color.blue)") // 50 50 50

u.color.red = 25

print("\(c.red) \(c.green) \(c.blue)")                    // 50 50 100
print("\(u.color.red) \(u.color.green) \(u.color.blue)") // 25 50 50
```

Int, Double, String, Array, Set, Dictionary

- Los tipos numéricos, String, Array, Set, Dictionary son estructuras.
  - Se copian al asignarlos a variables, o al pasarlos como parámetro de una función, etc…
  - En los arrays, sets y diccionarios, el proceso de copia está optimizado internamente y no se realiza hasta que no es absolutamente necesario.
    - No hay que preocuparse por la eficiencia.

Enumerations

Enumeration

- Una enumeración define un nuevo tipo formado por un grupo de valores excluyentes entre sí.
  - Es decir, nuestro dato solo puede ser uno de los definidos en la enumeración.
    - Solo podemos estar en primavera, verano, otoño o invierno.

```
enum Season {
    case Spring
    case Summer
    case Autumn, Winter // Pueden definirse varios
                        // valores en la misma línea
}
```

- Cada valor definido en la enumeración se llama miembro, y es un valor nuevo independiente.
  - No es como en C, donde cada valor es igual a un número entero.
- Las enumeraciones son un tipo manejado por valor.
  - Los valores se manejan por valor. Se asignan o pasan haciendo una copia del valor.

```
var s1: Season = Season.Summer
      // Defino una variable y la inicializo.

s1 = Season.Spring
      // Asigno otro valor a la variable.

s1 = .Autumn
      // Asigno otro valor a la variable.
      // Omito el nombre del tipo. Lo conozco.

var s2 = Season.Winter
      // Defino otra variable.
      // Omito el tipo - Se infiere.
```

```
switch s1 {    // switch exhaustivo con todos los casos.
case .Spring:                        // Puedo omitir Season
   print("Estamos en primavera.")
case .Summer:
   print("Estamos en verano.")
case .Autumn:
   print("Estamos en otoño.")
case .Winter:
   print("Estamos en invierno.")
}

switch s2 {   // switch exhaustivo con un default.
case .Spring:
   print("Estamos en primavera.")
default:
   print("No estamos en primavera.")
}
```

Valores Asociados

- Con cada valor miembro de un enumerado,
  - se pueden guardar valores asociados.
- El tipo de los valores asociados a cada valor miembro puede ser diferente.

```
enum Vehicle {
    case Car(String, Int)
    case Bicycle
    case Other(String)
}
```

- Las variables y constantes con el valor:
  - **`Vehicle.Car`** llevan asociados un **`String`** con la matrícula del coche, y un **`Int`** con el año de matriculación.
  - **`Vehicle.Bicycle`** no tienen valores asociados
  - **`Vehicle.Other`** llevan asociado un **`String`** describiendo el tipo de vehículo.

```swift
var v1: Vehicle = .Car("0000 AAA", 2010)

let v2 = Vehicle.Car("1111 AAA", 2014)

var v3 = Vehicle.Bicycle

var v4 = Vehicle.Other("Airplane")



switch v4 {                  // Extraer valores asociados con un switch
case .Car(let plate, var year):     // Uso var o let en cada valor
    year++
    print("Coche con matrícula \(plate) anterior al año \(year).")
case .Bicycle:
    print("Bicicleta")
case let .Other(descr):             // Uso un let global
    print("Es un \(descr).")
}


print(v4)     // Other("Airplane")
```

Raw Values

- Al definir un enumerado se puede asignar un valor raw a los miembros definidos
  - Los valores raw de todos los miembros son del mismo tipo.
    - Pueden ser String, Character, o números enteros o reales.

```
enum Season : Character { // Valores raw son Character
    case Spring = "p"
    case Summer = "v"
    case Autumn = "o"
    case Winter = "i"
}
```

- Si el tipo de los valores raw es entero o string, no hace falta especificar explícitamente el valor raw de cada miembro.
  - Si el tipo de los valores raw es entero, se asigna 0 como valor raw del primer miembro, y a los demás se les va sumando uno.

```
enum Day : Int {
    case Monday = 1     // valor raw es 1
    case Tuesday        // valor raw es 2
    case Wednesday      // valor raw es 3
    case Thursday = 50  // valor raw es 50
    case Friday         // valor raw es 51
    case Saturday       // valor raw es 52
    case Sunday         // valor raw es 53
}
```

  - Si el tipo de los valores raw es String, el valor raw de los miembros es el nombre del miembro.

```
enum Side : String {
    case Left    // valor raw es "Left"
    case Right   // valor raw es "Right"
}
```

- El método **`toRaw`** devuelve el valor raw de un miembro:

```
Season.Autumn.toRaw()   // "o"
Day.Friday.toRaw()      // 51
```

- El método de **`fromRaw`** busca el miembro de una enumeración asociado a un valor raw.
  - Nótese que es un método de Tipo.
  - Devuelve un tipo Optional,
    - ya que puede que no exista un miembro para el valor raw buscado.

```
if let s = Season.fromRaw("v") {
   s              // .Summer
} else {
   print("No existe miembro")
}
```

Enumeraciones Recursivas

- Una enumeración recursiva es aquella que se usa así misma para el valor asociado de uno (o varios) de sus miembros.
- Esto provoca que sea necesaria un reserva infinita de memoria para poder almacenar los valores enumerados.
- Se evita introduciendo un nivel de indirección.
- Añadiendo **indirect** delante de los cases donde exista recursividad, o delante de enum para que afecte a todos los cases.

```
enum Tree<T> {
    case Leaf(T)
    indirect case Node(Tree,Tree)
}

var a1: Tree<Int> = .Leaf(1)
var a2: Tree<Int> = .Leaf(2)
var a3: Tree<Int> = .Leaf(3)
var n1: Tree<Int> = .Node(a1,a2)
var n0: Tree<Int> = .Node(a3,n1)

n0   // Node(Leaf(3), Node(Leaf(1), Leaf(2)))
```

Propiedades

Propiedades

- Las propiedades contienen los valores de las clases, estructuras y enumerados.
- Las propiedades pueden ser:
  - **Almacenadas** (stored) o **calculadas** (computed).
  - De **instancia** o de **tipo**.
- Las propiedades pueden tener **observadores** que vigilan cuando se cambia su valor, para ejecutar ciertas acciones en ese momento.

Propiedades Almacenadas

- **Solo en clases y estructuras**.
  - Los enumerados no tienen propiedades almacenadas.
- Pueden ser variables o constantes dependiendo si su valor se puede cambiar o no..

```
struct Color {
   var red = 0
   var green = 0
   var blue = 0
   let alpha = 1
}
```

- Notas:
  - Si una instancia de una estructura es una contante, no puede cambiarse el valor de sus propiedades aunque sean var; dado que las estructuras son tipos manejados por valor.

```
let c = Color() // no puedo cambiar las propiedades de c
```

  - Con las clases no pasa esto al manejarse por referencia. Lo que es constante es la referencia, no lo apuntado por ella.

Propiedades Almacenadas Perezosas.

- Son propiedades almacenadas cuyo valor inicial no se calcula hasta que se usan por primera vez.
  - Deben declararse como variables (var)
  - Añadir **`lazy`** antes de la declaración.
- Se usan cuando:
  - No se tiene toda la información necesaria para asignarles el valor inicial.
  - El cálculo del valor inicial es costoso y solo se realiza cuando no queda más remedio.

```
class Message {
    var body = "Hola"
    lazy var attachments = load()
}

var m = Message()

m.body
m.attachments  // Ahora se calcula el valor de esta propiedad.
```

Propiedades Calculadas

- Soportadas por clases, estructuras y enumerados.
  - *La idea es similar a tener unos métodos `getCosa` y `setCosa`, que internamente sacan o ponen el valor de algún otro sitio, pero creados como una propiedad llamada `cosa`.*
- Hay que proporcionar un getter para obtener el valor de la propiedad.
  - y un setter (OPCIONAL) para salvar el valor de la propiedad.

```
struct Circle {
    var radius = 1.0        // Propiedad almacenada
    var area: Double {      // Propiedad calculada
        get {               // getter
            return M_PI * radius * radius
        }
        set(newArea) {      // setter
            radius = sqrt(newArea / M_PI)
        }
    }
}
var c = Circle()
c.radius                    // 1.0
c.area                      // 3.14159
c.area = 12.566371
c.radius                    // 2.0
```

- Puede omitirse el parametro en el setter.
  - Se le asigna por defecto el nombre **`newValue`**.

```
struct Circle {
    var radius = 1.0        // Propiedad almacenada
    var area: Double {      // Propiedad calculada
        get {               // getter
            return M_PI * radius * radius
        }
        set {               // setter
            radius = sqrt(newValue / M_PI)
        }
    }
}
```

- Propiedades Calculadas Read-Only.
  - Si se omite el setter, la propiedad es read-only.
  - Y puede simplificarse la declaración eliminando la palabra get y las llaves:

```
struct Circle {
    var radius = 1.0          // Propiedad almacenada
    var area: Double {        // Propiedad calculada
        return M_PI * radius * radius
    }
}
```

Observadores

- Los observadores de propiedades vigilan el valor de las propiedades y se ejecutan cada vez que se les asigna un valor.
  - Se ejecutan aunque el valor asignado sea el mismo que ya tenían.
  - Excepción: no se llaman cuando se inicializa una propiedad por primera vez.
- Pueden añadirse dos tipos de observadores:
  - **`willSet`**
    - Se llama antes de guardar el valor.
    - Toma como parámetro el nuevo valor.
      - Si no se escribe el nombre del parámetro, se crea por defecto con el nombre **newValue**.
  - **`didSet`**
    - Se llama después de guardar el valor.
    - Toma como parámetro el valor anterior.
      - Si no se escribe el nombre del parámetro, se crea por defecto con el nombre **oldValue**.
      - Desde este observador puede cambiarse el valor final asignado a la propiedad.
- Los observadores pueden añadirse a:
  - propiedades almacenadas (excepto perezosas)
  - propiedades heredadas (almacenadas y calculadas)
  - las variables globales y locales.

```
struct Circle {
    var radius: Double = 1.0 {
        willSet {
            print("Aumentara en \(newValue - radius)")
        }
        didSet {
            print("Aumento en \(radius - oldValue)")
        }
    }
}


var c = Circle()

c.radius = 2   // Aumentara en 1.0
               // Aumento en 1.0
```

Propiedades de Tipo

- Son propiedades que pertenecen al tipo, no a las instancias.
  - Se usan con notación punto sobre el nombre de tipo.
- Pueden ser almacenadas o calculadas.
  - Las almacenadas pueden ser constantes o variables.
    - Hay que darles siempre un valor inicial por defecto, ya que no existe un inicializador de Tipo.
    - El valor se asigna de forma perezosa.
      - Esta asignación se hace solo una vez y es Thread-Safe.
  - Las calculadas solo pueden ser variables.
- Se definen anteponiendo la palabra **`static`**.
- En las clases, las propiedades de tipo calculadas se definen usando la palabra **`class`**. para permitir que su implementacin pueda ser sobreescrita por sus subclase.

```
struct Color {
    var red = 0
    var green = 0
    var blue = 0

    static let black  = Color()

    static let white  = Color(red:255, green:255, blue:255)

    static let yellow = Color(red:255, green:255, blue:0)

}

let c = Color.yellow   // R:255 G:255 B:0
```

```
class Message {
    var header = "Hola"
    var body = "¿Qué hora es?"

    static var footer: String {
        return "(c) IWEB"
    }
}


Message.footer   // (c) IWEB
```

Métodos

Métodos

- Son funciones asociadas con un tipo.
  - Se definen con la misma sintaxis que las funciones.
- Pueden definirse métodos en las clases, en las estructuras y en los enumerados.
- Los métodos pueden ser de **Instancia** y de **Tipo**.

Métodos de Instancia

- Pertenecen a las instancias.
  - Se invocan con notación punto sobre una instancia.

```
class Position {
    var x = 0
    var y = 0
    func up() {
        y++
    }
    func upBy(steps:Int) {
        y += steps
    }
    func gotoOrigin() {
        x = 0; y = 0
    }
}
let p = Position()  // 0 0
p.up()              // 0 1
p.upBy(4)           // 0 5
p.gotoOrigin()      // 0 0
```

Nombres de los Parámetros

- Igual que en las funciones, los parámetros de los métodos pueden tener un nombre **interno** o **local** y un nombre **externo**,
  - El nombre del **primer parámetro** se considera solo **local**.
    - Se usa solo dentro del cuerpo del método.
  - Si se desea tener un nombre externo, añadirlo explícitamente.
  - El nombre de **los demás parámetros** se consideran **local** y **externo**.
    - Se usa dentro del cuerpo del método y en las llamadas al método.
  - Si se desea cambiar el nombre externo, añadirlo explícitamente; o incluso eliminarlo usando _.

```
class Position {
    var x = 0
    var y = 0

    // . . .

    func scaleBy(factor:Int, inHorizontal:Bool, inVertical:Bool) {
        if inHorizontal { x *= factor }
        if inVertical   { y *= factor }
    }
}                           // factor       - Nombre local
                            // inHorizontal - Nombre local y externo
                            // inVertical   - Nombre local y externo

let p = Position()

p.x = 3
p.y = 4

p.scaleBy(3, inHorizontal: false, inVertical: true)  // x=3 y=12
```

self

- **`self`** es una propiedad implícita existente en todas las instancias.
  - Es equivalente a la propia instancia.
- Se usa en los métodos de instancia para referirse a la propia instancia.
- En la práctica se usa poco.
  - Podemos omitir **self** al referirnos a propiedades o métodos de la propia instancia.
  - Es necesario usar **self** cuando el nombre de un parámetro del método tiene el mismo nombre que una propiedad de la instancia.

```
class Position {
    var x = 0
    var y = 0

    // . . .

    func gotoOrigin() {
        moveTo(x: 0, y: 0)      // No es necesario self.
    }

    func moveTo(#x:Int, y:Int) {
        self.x = x
        self.y = y
    }
}


let p = Position()

p.moveTo(x: 22, y: 33)  // x=22 y=33
p.gotoOrigin()          // x=0  y=0
```

Tipos Valor y Métodos Mutantes

- En los tipos Valor (**estructuras** y **enumerados**):
  - Los métodos de instancia no pueden modificar las propiedades.
    - Solo pueden modificarlos los métodos declarados como mutating.
      - Anteponer la palabra **mutating** al definir el método.
    - Nótese que si la instancia del tipo Valor es una constante *(es decir, no puede modificarse)*, entonces no puede usarse con un método mutante *(es decir, que pueda modificarla)*.
    - Si un método mutante asigna un nuevo valor a **self**, está cambiando el valor de la instancia por uno nuevo.

```
struct Size {
    var width = 0
    var height = 0

    mutating func scaleBy(factor:Int) {
        width  *= factor
        height *= factor
    }

    mutating func zero() {
        self = Size()        // Cambio la instancia por otra nueva
    }
}

var s1 = Size(width: 10, height: 20)
s1.scaleBy(2)  // width=20  height=40
s1.zero()      // width=0   height=0

let s2 = Size(width: 10, height: 20)
s2.scaleBy(2)  // ERROR: método mutating usado con constante
```

Métodos de Tipo

- Se invocan sobre el propio tipo, no sobre las instancias.
  - Se usan con notación punto sobre el nombre de tipo.
- Se definen anteponiendo la palabra **static**.
  - En la clases se antepone la palabra **class** cuando se permite que las subclases puedan sobreescribir el método.
- Dentro del cuerpo de un método de tipo, la propiedad **self** se refiere al tipo.

```
struct CashRegister {

    static var total = 0
    var subtotal = 0

    mutating func pay(money: Int) {
        CashRegister.total += money
        subtotal += money
    }

    static func balance() {
        print("Recaudación total = \(total)")
    }

     func subbalance() {
        print("Recaudación de la Caja = \(subtotal)")
    }

}

var cr1 = CashRegister()
var cr2 = CashRegister()
var cr3 = CashRegister()

cr1.pay(10)
cr2.pay(8)
cr3.pay(10)
cr2.pay(5)
cr1.pay(15)

cr1.subbalance()        // Recaudación de la caja = 25
cr2.subbalance()        // recaudación de la caja = 13
cr3.subbalance()        // Recaudación de la caja = 10

CashRegister.balance()  // Recaudación total = 48
```

Subscript

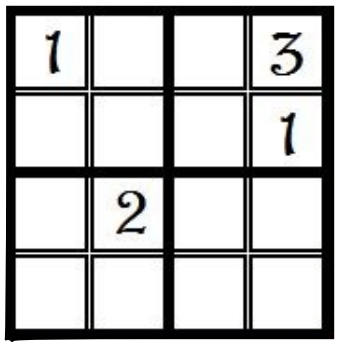
Subscript

- Es una sintaxis abreviada para acceder a los datos guardados en las enumeraciones, estructuras o clases, usando `[` y `]`.
  - Normalmente es para acceder a los datos guardados en colecciones, listas, secuencias, etc…
- La sintaxis para definirlos es una mezcla entre funciones y propiedades calculadas.

```
subscript(index: Int) -> Int {
    get {
        // devolver el valor correspondiente al índice dado
    }
    set(newValue) {
        // guardar el nuevo valor donde indique el indice
    }
}
```

- El subscript puede ser **readwrite** o **readonly**.
  - Si es read-only puede simplificarse la sintaxis igual que con las propiedades calculadas.
    - Eliminar la palabra get y las llaves.
- Si no se especifica el parámetro del setter, por defecto se llama **newValue**.
- El subscript puede tener varios parámetros de entrada, cada uno de un tipo diferente.
- Los parámetros pueden ser variables, pero no pueden ser in-out.
- Los parámetros pueden ser variadic, pero no pueden ser parámetros con un valor por defecto.
- El tipo de retorno del subscript puede ser de cualquier tipo.
- Los subscript se pueden **sobrecargar**:
  - Se pueden definir tantos subscripts como se desee.
  - Pero según los tipos de los parámetros o del retorno, deben poder diferenciarse para saber siempre cuál es el que hay que usar.

```
class Sudoku {
    var numbers = [Int](count: 16, repeatedValue: 0)

    subscript(box: Int, pos: Int) -> Int {
        get {
            return numbers[box * 4 + pos]
        }
        set(newValue) {
            numbers[box * 4 + pos] = newValue
        }
    }
    subscript(pos: Int) -> Int {
        return numbers[pos]
    }
}

var s = Sudoku()  // 0 0 0 0   0 0 0 0   0 0 0 0   0 0 0 0
s[1,2]            // 0
s[6]              // 0
s[0,0] = 1
s[1,1] = 3
s[1,3] = 1
s[2,1] = 2
s                 // 1 0 0 0   0 3 0 1   0 2 0 0   0 0 0 0
s[0,0]            // 1
s[0]              // 1
s[5]              // 3
s[1,3]            // 1
```



Herencia

Herencia

- La herencia solo aplica con **clases**.
- Herencia simple.
- No existe una clase base raíz.
  - Aunque algunas veces heredaremos de NSObject para tener algunas características especiales.
- Cualquier clase que no derive de otra es una clase base.
- Las subclases indican la clase padre después de su nombre separando con dos puntos.

```
class Animal {          // Clase base
}
class Dog : Animal {    // Clase derivada
}
var d = Dog()           // Crear instancia de perro
```

- La clase Dog hereda las propiedades, métodos y subscripts de la clase Animal.
  - Y puede sobreescribir los que desee modificar.

Sobreescribir

- Una clase hija puede sobreescribir los elementos heredados de su clase padre:
  - los métodos de clase e instancia,
  - las propiedades de clase y de instancia,
  - los subscripts.
- Es obligatorio anteponer la palabra **`override`** delante del elemento que se está sobreescribiendo.
  - Así se pueden detectar errores/despistes.
- Desde la clase hija se puede acceder a la versión de los métodos, propiedades o subscripts de la clase padre usando en prefijo **`super`**.
- Para evitar que algún elemento pueda sobreescribirse en una clase derivada debe marcarse con el modificador **`final`**.
  - Puede marcarse con **final** una clase entera, un método, una propiedad, un subscript.

```
class Animal {
    func eat() {
        print("El animal come")
    }
}

class Dog : Animal {
    override func eat() {
        super.eat()
        print("El perro come")
    }
}

var d = Dog()

d.eat()  // El animal come
         // El perro come
```

- Se pueden sobreescribir los getters y setter de las propiedades heredadas (almacenadas y calculadas).
- Es necesario indicar el nombre y el tipo de la propiedad (para saber que propiedad se está sobreescribiendo).
- Restricción: Una propiedad readwrite no puede sobreescribirse como una propiedad read-only.
  - Si se sobreescribe el setter de una propiedad, es obligatorio escribir también el getter.
- Pueden añadirse observadores a las propiedades heredadas.
  - Pero no se pueden añadir observadores en las propiedades heredadas que sean:
    - constantes y almacenadas.
    - read-only y calculadas.

Inicialización

Inicialización

- Preparar una instancia antes de que pueda usarse:
  - valores iniciales de las propiedades.
  - otras inicializaciones.
- Al crear una instancia de una Clase o Estructura, todas la propiedades almacenadas deben tener un valor inicial.
  - Ninguna propiedad almacenada puede quedarse sin un valor inicial.
  - El valor inicial se puede poner con un inicializador o usando valores por defecto para las propiedades.
- La forma más simple de inicializador es:

```
init() {
    // hacer aquí las inicializaciones
}
```

- En el inicializador pueden proporcionarse parámetros:

```
class Distance {
    var distanceInMeters: Double

    init() {
        distanceInMeters = 0
    }
    init(fromMeters meters: Double) {
        distanceInMeters = meters
    }
    init(fromKilometers kilometers: Double) {
        distanceInMeters = kilometers * 1000
    }
}

let d1 = Distance()                      // 0
let d2 = Distance(fromMeters:5)          // 5
let d3 = Distance(fromKilometers: 3)     // 3000
```

- Los parámetros de los inicializadores tienen un nombre **interno** y otro **externo**, igual que ocurre con los método y funciones.
  - Sin embargo, dado que el nombre del inicializador es siempre el mismo, el primer parámetro también tiene por defecto un nombre externo igual al nombre interno.
    - En la inicialización debe proporcionarse el nombre externo de **todos** los parámetros.
    - Aunque se puede proporcionar explícitamente un nombre externo diferente al interno, o usar _ para no usar un nombre externo.
- Las propiedades opcionales se inicializan automáticamente a **nil**.
  - Si el inicializador no proporciona un valor para ellas se quedan con nil.
- El inicializador también puede modificar el valor inicial de las propiedades constantes.

Inicializadores por Defecto

- En **Estructuras** y **Clases Base**:
  - Se proporciona automáticamente un inicializador por defecto para:
    - las estructuras y clases base que:
      - tienen un valor por defecto para todas sus propiedades,
      - y no proporcionan ningún inicializador propio.

    - El inicializador proporcionado por defecto crea las nuevas instancias con todas las propiedades inicializadas a su valor por defecto.

```
class Point {
    var x = 0
    var y = 0
}

var p = Point() // x=0 y=0
```

- En **Estructuras**:
  - Para las estructuras que no definen sus propios inicializadores,
    - Se crea automáticamente un inicializador **memberwise**.
  - Al inicializador memberwise se le pasa el valor inicial que deben tomar todas las propiedades de la nueva instancia.
    - En la llamada al inicializador, se pasa el valor de las propiedades junto con el nombre de las propiedades.

```
struct Color {
    var red = 0
    var green = 0
    var blue = 0
}

let c = Color(red:10, green:50, blue:200)
```

Delegación de Inicializadores en Tipos Valor

- **En estructuras y enumeraciones:**
  - Un inicializador puede delegar en otros inicializadores para realizar parte de la inicialización.
  - Un inicializador llama a otro inicializador con **self.init** y pasa los parámetros adecuados.
    - Nota: self.init solo puede llamarse desde un inicializador.

```
struct Color {
    var red = 0
    var green = 0
    var blue = 0
    init(red: Int, green: Int, blue: Int) {
        self.red = red; self.green = green; self.blue = blue
    }
    init(gray: Int) {
        self.init(red:gray, green:gray, blue:gray)
    }
}
let c1 = Color(red:10, green:50, blue:200) // R:10  G:50  B:200
let c2 = Color(gray:100)                   // R:100 G:100 B:100
```

Clases: Inicializadores Designated y Convenience

- En una clase pueden crearse dos tipos de inicializadores:
  - **Inicializadores Designados** (designated):
    - Son los inicializadores primarios.
      - Primero inicializan todas las propiedades introducidas en su propia clase.
      - Después llaman a un inicializador de su superclase inmediata para continuar con la inicialización (con `super.init(args)`).
        - El inicializador de la superclase realiza el mismo proceso.
      - Finalmente pueden modificar el valor de las propiedades heredadas o propias, invocar métodos de la clase, o usar self como un valor.
  - **Inicializadores de Conveniencia**:
    - Son inicializadores secundarios que se introducen por comodidad.
      - Primero invocan a un inicializador primario de la propia clase con **`self.init`**.
        - Un inicializador de conveniencia pueden invocar a otro inicializador de conveniencia, y éste a otro, pero al final de la cadena, el último inicializador de conveniencia debe llama a un inicializador designado.
      - Después pueden modificar el valor de las propiedades, tanto las suyas como las heredadas.
    - Se declaran con el modificador **`convenience`**.

```
class Point {
    var x: Double
    var y: Double

    init(x:Double, y:Double) {
        self.x = x
        self.y = y
    }
}

class Vector : Point {
    var angle: Double

    init(x:Double, y:Double, a:Double) {
        angle = a
        super.init(x:x, y:y)
    }

    convenience init(north: Double) {
        self.init(x:0, y:north, a:M_PI_2)
    }
}


let v1 = Vector(x:1, y:2, a:3)  // x=1.0 y=2.0 angle=3.0
let v2 = Vector(north:2)        // x=0.0 y=2.0 angle=1.57
```

Clases: Herencia y Sobreescritura

- Por defecto, una subclase no hereda los inicializadores de la superclase.
  - Solo se heredan en unos casos muy específicos:
    - Primero, para poder heredarlos, la subclase debe proporcionar un valor por defecto para todas las propiedades que introduce.
    - Si lo anterior se cumple, entonces:
      - Si la subclase no define ningún inicializador designado, entonces hereda los inicializadores designados de su superclase.
      - Si la subclase soporta todos los inicializadores designados de su superclase (porque los implementa o los ha heredado), entonces hereda todos los inicializadores de conveniencia de su superclase.
- Si una subclase sobreescribe un:
  - inicializador designado de su superclase, debe anotarlo con el modificador **`override`**.
  - inicializador de conveniencia de su superclase, **NO** debe anotarlo con ~~**`override`**~~.

Clases: Inicializadores Requeridos

- Si en una clase se marca un inicializador con el modificador **`required`**, entonces:
  - todas las clases derivadas de esta deben implementar ese inicializador,
    - y además deben anotarlo también con **`required`**.
    - Nota: en este caso no es necesario usar también **`override`**.
  - Pero la subclase puede no necesitar implementar un inicializador requerido si cumple las condiciones para heredarlo.

Inicializar Propiedad con Closure o FG

- A una propiedad almacenada se le puede asignar un valor por defecto que sea el resultado de ejecutar un closure o una función global.
  - El crear una instancia, se ejecuta el closure o la función global, y el valor devuelto es el valor por defecto asignado a la propiedad.
  - Téngase en cuenta que mientras se ejecuta el closure o la función global, la instancia no está completamente inicializada, y por tanto, no puede accederse a otras propiedades, métodos, o al valor self.

```
class Speed {
    var speed: Double = {
        // sentencias
        return 100.0
    }()
}

let s = Speed()   // s.speed = 100.0
```

Inicializador *Failable*

- Es un inicializador que se usa cuando la creación de la instancia puede fallar.
  - es decir, la instancia puede no crearse.
- Llevan una `?` después de `init`.
- Crean un valor **Optional** del tipo que inicializan.
  - Si no se puede crear la instancia, ejecutan `return nil`.
- Si se pone `!` después de `init`, el inicializador devuelve un valor **Implicitly Unwrapped Optional** del tipo que inicializan.

```
class Triangle {
    var a:Double, b:Double, c:Double

    init?(a:Double, b:Double, c:Double) {
        self.a = a
        self.b = b
        self.c = c
        if a+b<c || a+c<b || b+c<a {return nil}
    }
}

var c1 = Triangle(a:10, b:12, c:14)     // c1! es un Triangle
print("\(c1!.a) \(c1!.b) \(c1!.c)")   // 10.0 12.0 14.0

var c2 = Triangle(a:4,  b:3,  c:10)     // c2 es nil

if let c3 = Triangle(a:10, b:4, c:3) {  // c3 es un Triangle
   print("\(c3.a) \(c3.b) \(c3.c)")   // No se cumple el if
}
```

- **¿Cuándo podemos hacer que falle el inicializador?**
  - En los tipos valor (enumeraciones y estructuras):
    - Dentro la implementación de un inicializador failable, podemos llamar a return nil en cualquier punto.
  - En las clases:
    - Solo podemos hacer fallar a un inicializador failable cuando a todas las propiedades almacenadas introducidas por la clase se les ha dado un valor inicial, y después haber llamado a los inicializadores en los que se esté delegando.
- **Inicializadores failables creados por defecto:**
  - En las enumeraciones se crea automáticamente un inicializador llamado `init?(rawValue:)` para crear una instancia de la enumeración a partir de un valor **raw**.

- **¿Cuando puede un inicializador delegar en otro inicializador?**
  - Un inicializador failable puede delegar en otro inicializador failable.
    - Si el inicializador en el que se ha delegado falla, el proceso de inicialización total falla inmediatamente, y no se ejecutan más sentencia de inicialización.
  - Un inicializador failable también puede delegar en un inicializador no failable.
  - Un inicializador no failable nunca puede delegar en un inicializador failable.

- **Sobreescribir un inicializador failable:**
  - Una subclase puede sobreescribir un inicializador failable de una superclase.
    - con un inicializador failable o también con uno no failable.

Des-Inicialización

Deinitializer

- **Solo para Clases.**
- Se usan típicamente para liberar algunos recursos propios creados por el desarrollador.
  - Pero no para liberar la memoria: Swift gestiona la memoria con ARC.
- Se llaman automáticamente justo antes de liberar la memoria de una instancia de clase.
  - El desarrollador nunca debe llamarlos en su código.
- Una clase solo puede tener un desinicializador, o ninguno.
- Se definen con la palabra **deinit**, seguido de cuerpo a ejecutar.
  - No tienen parámetros.

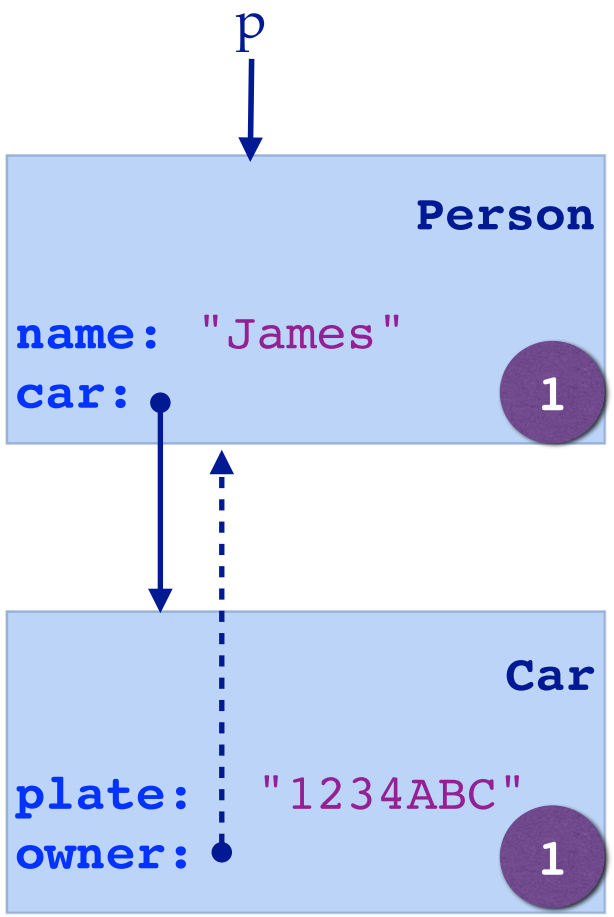
```
class Name {
    deinit {
        // sentencias
    }
}
```

Automatic Reference Counting
ARC

Clases: ARC

- La cuenta de referencias solo afecta a las instancias de clases.
- El contador de referencias de una instancia indica cuantos elementos la retienen.
  - La memoria de la instancia no se libera mientras el contador sea mayor que cero.
- Tipos de referencias:
  - **`strong`**:
    - Una referencia strong aumenta en uno el contador de referencias de la instancia apuntada.
    - Las referencias **por defecto** son siempre strong.
  - **`weak`**:
    - Estas referencias no aumentan el contador de referencias de las instancias apuntadas.
    - Una referencia weak se pone a **nil** si se libera la memoria de la instancia apuntada.
      - Por tanto, las referencias weak deben ser **variables** y declararse como **Tipos Optional**.
  - **`unowned`**:
    - Estas referencias no aumentan el contador de referencias de las instancias apuntadas.
    - Se usan cuando se sabe que la instancia apuntada **nunca** va a liberarse.
      - Mas cómodo que weak al no tener que desempaquetar los valores referenciados.
      - Pero si se libera el objeto apuntado, se produce un runtime error.
- Las referencias weak y unowned se usan para evitar crear **bucles de retenciones**.

```
class Person {
    let name: String
    var car: Car?

    init(name: String) {
        self.name = name
    }
}

class Car {
    let plate: String
    weak var owner: Person?

    init(plate: String, owner: Person) {
        self.plate = plate
        self.owner = owner
    }
}

var p = Person(name:"James")
p.car = Car(plate: "1234ABC", owner: p)

if let car = p.car {
    print("El coche de \(p.name) es \(car.plate).")
} else {
    print("\(p.name) no tiene coche.")
}
```



Bucles Strong con Closures

- Preliminares:
  - Las closures son Tipos Referencia.
    - y por tanto pueden ser retenidas.
  - Una closure también puede retener (capturar) otras instancias
    - si en el cuerpo de la closure se accede a una propiedad de la instancia, o si llama a un método de la instancia.
- Por tanto, también es posible crear bucles de retenciones strong al usar closures.
  - Ejemplo:
    - Una instancia "i" guarda una closure "c" en su propiedad "p".
      - es decir i.p retiene a c.
    - Y en el cuerpo de la closure "c" se accede al método "m" de "i".
      - es decir, la closure esta capturando/reteniendo a la instancia "i".
    - Ya tenemos el bucle de retenciones strong: "i" y "c" se retienen mutuamente.
- Se soluciona usando **listas de capturas** en las closures.

```
class MultiplicationTable {

    let number: Int

    lazy var calculate: (Int) -> Int = {
       (n:Int) -> Int in
           self.number * n
    }

    func multiplyBy(n:Int) -> Int {
        return calculate(n)
    }

    init(number: Int) {
        self.number = number
    }

    deinit {
        print("Desinicializando")
    }
}

if true {
   var t = MultiplicationTable(number: 7)
   t.multiplyBy(3)  // 21
}
```

calculate retiene a la closure encargada de multiplicar **self.number** por el valor pasado como parámetro.

Y la closure retiene a **self** dado que usa **self.number**.

Ya tenemos el bucle de retenciones strong: la instancia retiene a la closure y ésta retiene a la instancia.

Nunca aparece por consola el mensaje de deinit.

Closures: Lista de Capturas

- Las listas de Capturas de las closures se usan para evitar crear bucles de retenciones strong.
- Se añaden en la definición de las closures:
  - declaran si las referencias capturadas por la closure deben ser **weak** o **unowned**, en vez de **strong**.
  - Sintaxis:
    - Es una lista de parejas separadas por comas y entre corchetes situada después del abre-llaves y antes de los parámetros.
    - Cada pareja se forma con la palabra weak o unowned, y el nombre de la referencia capturada.
    - Ejemplo:

```
{[unowned self, weak p] (i:Int) -> String in . . . }
```

```
class MultiplicationTable {

    let number: Int

    lazy var calculate: (Int) -> Int = {
       [unowned self]
         (n:Int) -> Int in
             self.number * n
    }

    func multiplyBy(n:Int) -> Int {
        return calculate(n)
    }

    init(number: Int) {
        self.number = number
    }

    deinit {
        print("Desinicializando")
    }
}

if true {
   var t = MultiplicationTable(number: 7)
   t.multiplyBy(3)  // 21
}
```

La closure no necesita retener a **`self`**.

La instancia y la closure siempre se van a referenciar entre ellas, y se destruirán a la vez.

Ninguna sobreviva a la otra.

Después del if, var se destruye y aparece por consola el mensaje de deinit.

Manejo de Errores

Manejo de Errores

- Cómo manejar y recuperarse de situaciones de error que se producen en tiempo de ejecución.
  - cómo se capturan los errores, se lanzan, se propagan, se crean, …
  - cómo se definen tipos de errores personalizados, …
- Nótese que:
  - Algunos casos de error pueden manejarse simplemente usando Optionals:
    - Cuando no interese conocer los detalles sobre los errores producidos.
  - Hay situaciones de error que son irrecuperables, y lo mejor es que el programa muera.
- El manejo de errores en Swift es compatible con los mecanismos de gestión de errores y el uso de NSError en Cocoa y Objective-C.

Protocolo **ErrorType**

- Los tipos usados para representar errores deben ser conformes con el protocolo **ErrorType**.
- Los tipos de error se suelen implementar típicamente como enumerados conformes con ErrorType,
  - con varios cases para los posibles errores,
  - y con valores asociados para los cases que necesiten usar información adicional sobre el error.

```
enum UserError: ErrorType {
    case UnkownUserError
    case NoAuthorizedUserError(code: Int)
}
```

Lanzar Errores: **throw**

- Los errores se lanzan con **throw**

```
throw UserError.NoAuthorizedUserError(code: 5)
```

- Al lanzar un error, el flujo de ejecución normal del programa se detiene.
  - Alrededor de punto donde se lanzo el error, existirá algún elemento preparado para capturar el error y manejar la situación.
    - Solucionando el problema, intentando hacer algo alternativo, informando del problema, …

¿Cómo Manejar los Errores?

- Hay varias formas de manejar los errores que pueden aparecer en los distintos puntos de un programa:
  - Los errores que pueden producirse dentro de una función, pueden propagarse hacia el punto que invocó la función.
  - Manejar los errores que se produzcan con sentencias do-catch.
  - Manejarlos convirtiéndolos en valores Optionals.
  - Indicando que aunque teóricamente es posible que se produzcan un error, indicar que sabemos que esto no va a ocurrir, y de esta forma no hacer ningún tipo de manejo.

Sentencia **try**

- Cuando una función lanza una excepción, se altera el flujo normal de ejecución del programa.
- Swing requiere que estos puntos se identifiquen claramente para que el código se entienda mejor.
- Para ello las llamadas a funciones que puedan lanzar errores deben anteponer **`try`**, **`try?`** o **`try!`**.
  - En las llamadas a función, métodos e inicializadores.

```
try funcion1()
try! funcion2()
var a = try funcion3()
if let a = try? funcion4() { … }
```

Funciones Throwing

- Las funciones, métodos e inicializadores que pueden lanzar errores deben marcarse con **`throws`**.

```
func demo() throws -> String {
    . . .
    throw UserError.UnknownUserError
    . . .
}
```

- Cuando se produce o se lanza un error dentro de una función Throwing, este se propaga hasta el punto donde se invocó la función.
- Los errores que se producen dentro de una función no Throwing deben manejarse dentro de la función: no se propagan.

Sentencia **do-catch**

- Esta sentencia se usa para capturar los errores que se producen en las sentencias del do,
  - y se comparan con los catch's existentes para identificar cual debe manejar el error.
- Esta sentencia tiene este aspecto:

```
do {
   sentencias
   try expresion
   sentencias
} catch patron {
   sentencias
} catch patron where condicion {
   sentencias
} catch {
   sentencias
}
```

- El error que se lanza se compara en orden con los patrones de los catch para ver con cuál encaja, y ese catch manejará el error.
- En los catch con una condición **where**, también se comprobara si esta se cumple.
- Un catch sin patrón encajará con todos los errores. En este caso se crea una constante llamada **error** que contiene el error que se ha producido.
- Si ningún catch encaja con el error, entonces este sigue propagándose.

Convertir Errores en Opcionales

- **`try?`** se usa para manejar/descartar los errores lanzados por una función, y convirtiendo el valor devuelto por la función en un Optional.

```
let a = try? funcion()
if let b = try? funcion() { … }
```

- Si la función lanza un error, **try?** lo transfoma en **nil**.
- Si la función devuelve valores de un tipo T, al usar try?, los valores devueltos son Optional de T.

Detener la Propagación de Errores

- Si sabemos que una función Throwing *(que podría potencialmente lanzar un error)* no va a lanzar nigún error, entonces podemos llamarla con **`try!`**.
  - Esto deshabilita la propagación de errores.
  - Pero si se lanza un error, el programa de detendría con un error de ejecución.

```
var img = try! loadImage(path)
  // Estamos seguros de que la imagen
  // está disponible, que se cargará
  // sin problemas, sin que se produzca
  // ningún error.
```

Type Casting

Clases: Type Casting

- Type Casting solo es para clases.
- Se usa para:
  - Comprobar si una instancia es de un determinado tipo.
    - El operador para comprobar tipos es **`is`**.
      - Devuelve un booleano.
  - Usar una instancia como si su tipo fuera el de una de sus superclases o subclases.
    - Los operadores para cambiar de tipo son **`as`** , **`as?`** y **`as!`**.
      - **`as`** para cambiar el tipo a una clase base (**upcasting**).
        - Se comprueba al compilar y no puede fallar en ejecución.
      - **`as?`** para cambiar el tipo a una clase derivada (**downcasting**).
        - Usarlo si el casting puede fallar.
        - Devuelve un Optional.
      - **`as!`** para cambiar el tipo a una clase derivada (**downcasting**).
        - Es la forma forzada.
        - Se usar cuando se sabe que el casting va a tener éxito.

```
class Animal {
    // . . .
}

class Dog : Animal {
    // . . .
}

class Bird : Animal {
    // . . .
}


var animals: [Animal] = … // array con objetos Dog y Bird.


for a in animals {
    if a is Dog {
        print("Es un perro")
    } else if a is Bird {
        print("Es un pájaro")
    }
}
```

```
var ad : Animal = Dog()       // De tipo Animal, pero apunta a un perro.
var ab : Animal = Bird()      // De tipo Animal, pero apunta a un pájaro.

var d1 : Dog = ad as! Dog     // Downcasting forzado con éxito.
var d2 : Dog = ab as! Dog     // Runtime error:
                              // Falla el downcasting a Perro.

var d3 : Dog? = ad as? Dog    // Downcasting con éxito. d3! es el perro.
var d4 : Dog? = ab as? Dog    // Downcasting fallido.   d4 es nil.


if let d = ad as? Dog {
    print("Es un perro.")      // Es un perro.
} else {
    print("No es un perro.")
}

if let d = ab as? Dog {
    print("Es un perro.")
} else {
    print("No es un perro.")    // No es un perro.
}

var a:Animal = d1 as Animal       // Upcasting con éxito.
                                  // Se comprueba al compilar.
```

- En un switch puede usarse **`is`** y **`as`**.
  - Pero no pueden usarse **`as?`** ni **`as!`**.

```
switch a {
case let d as Dog:
    print("Es un perro llamado \(d.name)")
case let b as Bird:
    print("Es un pájaro llamado \(b.name)")
case is Dog:
    print("Es un perro que no me interesa")
case is Bird:
    print("Es un pájaro que no me interesa")
default:
    print("Es otra cosa")
}
```

Any y AnyObject

- **`Any`**
  - Representa una instancia de cualquier tipo.
    - incluidos tipos que son Clases.
- **`AnyObject`**
  - Representa instancias de cualquier tipo de Clase.

Tipos Anidados

Tipos Anidados

- Se pueden definir tipos anidados dentro de otros tipos.
  - Definiendo el tipo anidados dentro de las llaves de la definición del tipo que lo contiene.

- Para usar un tipo anidado fuera del contexto donde se definió,
  - usar el nombre del tipo contenedor como prefijo de su nombre.

```
class Chessboard {
    enum Shape { case Pawn, Rook, Knight, Bishop, Queen, King }
    enum Color { case White, Black }
    struct Piece {
        let shape: Shape
        let color: Color
    }
    var board = Array<Piece?>(count: 64, repeatedValue: nil)
    init() {
        board[0] = Piece(shape: .Rook,   color: .White)
        board[1] = Piece(shape: .Knight, color: .White)
        board[8] = Piece(shape: .Pawn,   color: .White)
    }
    subscript(index: Int) -> Piece? {
        get { return self.board[index] }
        set { self.board[index] = newValue }
    }
}
var cb = Chessboard()

cb.board[2] = Chessboard.Piece(shape: .Bishop, color: .White)
cb[3]       = Chessboard.Piece(shape: .Queen,  color: .White)

if let color = cb[3]?.color {
    if color == Chessboard.Color.White {
       print("Pieza blanca")             // Pieza blanca
    } else {
       print("Pieza negra")
    }
}
```

Extensiones

Extensions

- Permiten añadir nuevas funcionalidad a tipos ya existentes (clases, estructuras y enumeraciones):
  - Añadir propiedades calculadas de instancia y de tipo.
  - Definir nuevos métodos de instancia y de tipo.
  - Añadir nuevos inicializadores.
  - Definir subscripts.
  - Definir y usar nuevos tipos anidados.
  - Hacer que un tipo ya existente sea conforme a un protocolo.
- No permiten sobreescribir funcionalidades ya existentes.
- Pueden extenderse también tipos de los que no se tenga acceso al código fuente.

- Se declaran usando la palabra extension:

```
extension UnTipo {
    // nuevas funcionalidades
}

extension UnTipo: Protocolo1, Protocolo2 {
    // Implementación de los protocolos
}
```

- Una vez definida una extensión para un tipo, todas las instancias de ese tipo tienen disponible todas las funcionalidades añadidas,
  - aunque las instancias se crearán antes de definir la extensión

```
extension Int {
    var odd: Bool { return self % 2 != 0 }
    init(base: Int, exponent: Int) {
        self = base * Int(pow(10.0, Double(exponent)))
    }
    func repeat(task: () -> ()) {
        if self < 0 { return }
        for _ in 0..<self {
            task()
        }
    }
    mutating func double() {
        self *= 2
    }
    subscript(index: Int) -> Int {
        var p: Int = self
        abs(index).repeat { p /= 10 }
        return p % 10
    }
}

4.odd                                   // false
3.repeat {print("hola")}              // hola hola hola
var x = Int(base: 3, exponent: 2)   // 300
x.double()                              // 600
12345[3]                                // 2
```

```swift
extension Int {
    var sign: Sign { return self < 0 ? .Negative : .Positive }
    enum Sign { case Negative, Positive }
}

4.sign                  // Int.Sign.Positive



protocol Logable {
    func printLog(msg: String) -> ()
}

extension Int : Logable {
    func printLog(msg: String) -> () {
        print("\(msg) = \(self)")
    }
}

5.printLog("Primero")   // Primero = 5
4.printLog("Segundo")   // Segundo = 4
```

Protocolos

Protocolos

- Un protocolo define cuáles son los métodos, propiedades, inicializadores o requisitos necesarios para realizar una tarea o soportar alguna funcionalidad.
  - El protocolo **NO** proporciona una implementación,
  - Solo describe cómo debe ser la implementación.
- Los protocolos son **adoptados** por clases, estructuras y enumeraciones,
  - que son los que implementan la funcionalidad descrita por el protocolo.
- Cuando un tipo satisface los requisitos descritos por un protocolo, se dice que es **conforme** con ese protocolo.
- Los protocolos pueden usarse igual que cualquier otro tipo:
  - Puede usarse como el tipo de propiedades, el tipo parámetros de métodos, tipo de retorno de funciones, el tipo de Arrays, etc. . .

- Sintaxis para definir un protocolo:

```
protocol UnProtocolo {
    // Elementos definidos por el protocolo
}
```

- El tipo que adopta un protocolo:

```
struct UnTipo: UnProtocolo, OtroProtocolo {
   // Elementos del Tipo
}
```

- Si el tipo que adopta un protocolo también deriva de una superclase:

```
class ClaseHija: ClasePadre, UnProtocolo, OtroProtocolo {
    // Elementos del Tipo
}
```

Protocolos: Propiedades

- Un protocolo puede requerir que los tipos conformes con él proporcionen:
  - propiedades de instancia y de tipo,
  - e indicar si se deben ser proporcionar acceso get, o get y set.
  - Pero no especifica si deben ser almacenadas o calculadas.
- Para definir las propiedades:
  - Definirlas siempre como variables (**`var`**).
  - Indicando **`{get set}`** o **`{get}`** según el tipo uso requerido.
  - Si la propiedad es de tipo, usar siempre el prefijo **`class`**.

```
protocol UnProtocolo {
    var unaPropiedad : UnTipo {get}
    class var otraPropiedad : OtroTipo {get set}
}
```

Protocolos: Métodos

- Un protocolo puede requerir que los tipos conformes con él implementen:
  - métodos de instancia y de tipo.
    - Los métodos pueden tener parámetros variadic, pero no parámetros con valores por defecto.
- Para definir los métodos:
  - Especificar la cabecera de los métodos, no su cuerpo.
  - Los métodos de clase usan siempre el prefijo **`class`**.
  - Si el método es mutante, usar el prefijo **`mutating`**.

```
protocol UnProtocolo {
    func unMetodo(a: Int) -> Bool
    class func otroMetodo(b: Bool)
    mutating func otroMetodoMas(x: Double) -> Double
}
```

Protocolos: Inicializadores

- Un protocolo puede requerir que los tipos conformes con él implementen inicializadores.

```
protocol UnProtocolo {
    init()
}
```

- Las clases conformes deben usar el modificador **required** con estos inicializadores para obligar a que se implementen en todas las subclases.
  - Excepto si se han marcado con **final** (las subclases no los pueden sobreescribir).
- Si la implementación del inicializador requerido por el protocolo, además sobreescribe a algún inicializador designado de una superclase, deberá llevar también el modificador **override**.
- Si el inicializador requerido por el protocolo es ***failable***, puede implementarse como un inicializador ***failable*** o **no** ***failable***.
- Si el inicializador requerido por el protocolo es ***no failable***, puede implementarse como un inicializador ***no failable***, o con un inicializador ***implicitly unwrapped failable***.

Protocolos: Extensiones

- Se puede extender un tipo ya existente para que sea conforme con un protocolo.

```
extension UnTipo: UnProtocolo {
    // Implementar los elementos del protocolo
}
```

  - En el cuerpo de la extensión hay que implementar todo lo necesario para que el tipo sea conforme con el protocolo.
    - Caso extremo: Si el tipo ya implementa todos los requisitos especificados por el protocolo, no habría que añadir nada nuevo en el cuerpo de la extensión.

Protocolos: Herencia

- Un protocolo puede heredar de otros protocolos:

```
protocol UnProtocolo: Protocolo1, Protocolo2 {
    // Elementos definidos por el protocolo
}
```

  - Los tipos que adopten este protocolo deben ser conformes con los requisitos definidos en él y en todos los protocolos heredados.

Protocolos Class-Only

- Puede limitarse la adopción de un protocolo solamente a clases.
  - es decir, impedir que pueda adoptarse por enumeraciones o estructuras.
- Añadiendo **class** como el primer elemento en la lista de herencia:

```
protocol UnProtocolo: class, OtroProtocoloHeredado {
  . . .
}
```

- Se usa cuando los requisitos especificados en el protocolo asumen o requieren que el tipo que lo adopte funcione por referencia, no por valor.

Protocolos: Composición

- Para indicar que una propiedad, un parámetro, un valor de retorno, etc. debe ser conforme con varios protocolos, se usa la **composición de protocolos**.
- Se indica con la palabra **protocol** seguida de los nombres de los protocolos entre **<** y **>**.

```
var x: protocol<UnProtocolo, OtroProtocolo>

func unafuncion(v: protocol<UnProtocolo, OtroProtocolo>) {
}

func otraFuncion() -> protocol<UnProtocolo, OtroProtocolo> {
    return ???
}
```

- Una composición de protocolos es otro tipo, y puede usarse igual que cualquier otro tipo.

Protocolos: **is** y **as**

- Para comprobar si una instancia es conforme con un protocolo se usa el operador **`is`**.
- Para realizar un casting a un determinado protocolo se usa el operador **`as`**.
  - La versión Optional es **`as?`**, que devuelve un Optional del tipo del protocolo.

```
@objc protocol Domesticable { . . . }
class Animal {}
class Dog : Animal, Domesticable { . . .}
class Lion : Animal {}

var a : Animal = ???

if a is Domesticable { . . . }

if let d = a as? Domesticable { . . .}
```

  - Notas:
    - Para usar **`is`** es necesario marcar los protocolos con el atributo **`@objc`**.
    - Los protocolos marcados con **`@objc`** solo pueden ser adoptados por clases.

Protocolos: Requisitos Opcionales

- Los requisitos opcionales no tienen que implementarse obligatoriamente por los tipos que adopten el protocolo.
- Los requisitos opcionales se indican con el prefijo **`optional`**.
- Los requisitos opcionales (*propiedades y métodos con un valor de retorno*) devuelven un valor Optional al acceder a ellos, dado que pueden no haber sido implementados.
  - Usarlos con Optional Chaining cuando se usen en cadenas de llamadas.

```
@objc protocol Domesticable {
    optional var growl: String {get}
    optional func talk() -> String
}
class Dog : Domesticable {}

var dom: Domesticable = ???

if let growl = dom.growl {
    dom.talk?()
}
```

- Nota:
  - Los requisitos opcionales solo pueden usarse en protocolos marcados en el atributo **`@objc`**.
  - Los protocolos marcados con **`@objc`** solo pueden ser adoptados por clases.

Extensión de Protocolos

- Los protocolos pueden extenderse para proporcionar implementaciones por defecto de métodos y propiedades.
  - Así no es necesario escribir una implementación independiente en cada uno de los tipos que adopten el protocolo.
    - Pero cualquier tipo puede sobreescribir la implementación por defecto proporcionada por el protocolo.

- Creamos una extensión del protocolo **CollectionType** para añadir el método **countIf** con una implementación por defecto:

```
extension CollectionType {

    func countIf(condition: Generator.Element -> Bool)
                                        -> Int {
        var n = 0
        for value in self where condition(value) {
            n++
        }
        return n
    }
}

var c = [1,2,3,4,5,6,7].countIf({$0 > 3})  // 4
```

- Los tipos conformes con este protocolo (Array, Set, …) tienen ahora este método.

- **Añadir restricciones:**
  - Pueden ponerse restricciones en las extensiones de protocolos usando la cláusula **where**.
  - Los tipos que adopten el protocolo deben satisfacer las restricciones para que los métodos y propiedades definidos en la extensión estén disponibles.

```
extension CollectionType
    where Generator.Element: CustomStringConvertible {

    var textualDescription: String {
            return "[" + self.map({$0.description})
                              .joinWithSeparator(", ") + "]"
    }

}
```

Genéricos

Genéricos

- Los genéricos permiten crear funciones y tipos que pueden trabajar con cualquier tipo.
  - Ejemplos:
    - El tipo **Array** permite trabajar (almacenar) cualquier otro tipo.

```
var a = Array<Int>()  // Array de enteros
var b = Array<Dog>()  // Array de Perros
```

    - La función global **swap** permite intercambiar el valor dos variables de cualquier tipo.

```
var a: Int, b: Int, c: Dog, d: Dog
swap(&a, &b)  // Intercambio con variables de tipo Int
swap(&c, &d)  // Intercambio con variables de tipo Dog
```

      - Nota: La función **swap** está definida como:

```
func swap<T>(inout a: T, inout b: T)
```

Funciones Genéricas

- Las funciones genéricas funcionan con cualquier tipo.
- Indicar después del nombre de la función los nombres de los tipos comodines, separados por comas, y entre < y >.
  - Los nombres comodines se usarán para indicar el tipo de los parámetros, y en el cuerpo de la función.

```
func swap<T>(inout a: T, inout b: T) {
    (a, b) = (b, a)
}
```

  - Cuando se invoque la función genérica se determinará cual es el tipo real a usar en lugar de **T**.

```
var a: Int = 1, var b: Int = 2
swap(&a, &b)    // T es Int
```

Tipos Genéricos

- Son clases, estructuras o enumeraciones que trabajan con cualquier tipo.
- Al definir el tipo: indicar después de su nombre, los nombres de los tipos comodines separados por comas, y entre < y >.

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

- Al usar el tipo genérico: proporcionar los tipos reales a usar después del nombre del tipo, también entre < y >.

```
var s = Stack<Int>()
s.push(1)
s.push(2)
s.push(3)
var x = s.pop() // 3
```

- Ejemplo: un enumerado genérico para representar una alternativa. Cada alternativa lleva un valor asociado de distinto tipo.

```
enum Either<T1,T2> {
    case First(T1)
    case Second(T2)
}

var a: Either<String,Int>

a = .First("hola")
a = .Second(22)
```

  - NOTA: ESTO NO SE PODIA HACER EN SWIFT 1.2

Enumeraciones Recursivas

Extender Tipos Genéricos

- Al extender un tipo genérico no hay que proporcionar la lista de tipos comodines, se usa la del tipo original.

```
extension Stack {
    func top() -> T? {
        return items.last
    }
}

s.top()  // 2
```

Restricciones

- Pueden imponerse restricciones sobre los tipos que pueden usarse con una función genérica o con un tipo genérico.
- Las restricciones pueden especificar que:
  - los tipos sean subclases de otra clase,
  - sean conformes a un protocolo o composición de protocolos.
- Ejemplo:

```
func unaFuncion<T:UnaClase,U:UnProtocolo>(a:T,b:U) {}
```

  - Esta función genérica requiere que el tipo del parámetro **a** sea derive de ***UnaClase***, y que el parámetro **b** sea conforme al protocolo ***UnProtocolo***.

Tipos Asociados

- Al definir un protocolo no pueden usarse genéricos tal y como los hemos visto hasta ahora.
  - Pero pueden usarse **tipos asociados** (nombres de tipo comodines) usando **`typealias`** para realizar la misma tarea.
- ¿Cómo funciona?:
  - Al definir un protocolo se usa **`typealias`** para declarar nombres de tipos asociados.
  - Cuando un tipo a adopta el protocolo, se usa **`typealias`** para indicar cual es el tipo real que hay que usar para los tipos asociados.

```
protocol Queue {
    typealias T
    func get() -> T?
    func put(item: T)
}

struct Task {
    var name: String
}

class Dispatcher: Queue {
    typealias T = Task
    var tasks = [Task]()
    func get() -> Task? { return tasks.removeLast() }
    func put(task: Task) { tasks.insert(task, atIndex: 0) }
}

var d = Dispatcher()
d.put(Task(name: "Tarea 1"))
d.put(Task(name: "Tarea 2"))
d.put(Task(name: "Tarea 3"))
d.get()   // Tarea 1
d.get()   // Tarea 2
d.get()   // Tarea 3
```

Cláusulas **where**

- Pueden definirse requisitos sobre los tipos asociados de un protocolo usando cláusulas **`where`**.
  - Puede requerirse que:
    - un tipo asociado sea conforme a un protocolo.
    - un tipo asociado sea de un determinado tipo.

```
func transfer<Q1: Queue, Q2: Queue
                  where Q1.T == Q2.T>(q1: Q1, q2: Q2) {
    if let x = q1.get() {
        q2.put(x)
    }
}
var d1 = Dispatcher(), d2 = Dispatcher()
transfer(d1,d2)
```

  - La función **transfer** es una función genérica que toma como argumentos dos instancias conformes con **Queue**.
    - La cláusula **where** impone que el tipo asociado de ambas **Queue** sea el mismo.
    - Es decir esta función pasa un elemento entre colas que contienen el mismo tipo de elementos.

Control de Acceso

Control de Acceso

- Restringir el acceso a ciertas partes del código desde otros sitios.
  - Restringir el acceso a tipos, variables, constantes, funciones, propiedades, …
- Tipos de control de Acceso:
  - **Acceso público**: Se puede acceder a los elementos con el modificador **`public`** desde cualquier fichero y desde otros módulos.
  - **Acceso interno**: Se puede acceder a los elementos con el modificador **`internal`** desde cualquier fichero del propio módulo, pero no desde otros módulos.
    - Por defecto, si no se indica nada, todos los elementos tienen un control de acceso interno,
      - aunque hay algunas excepciones.
  - **Acceso privado**: Se puede acceder a los elementos con el modificador **`private`** solo desde el mismo fichero en el que están definidos. No se pueden usar desde otros ficheros ni desde otros módulos.

```
public class Dog {…}

internal let speed = 0

private func run() {…}

var total = 33    // por defecto es internal
```

- El nivel de acceso de los diferentes elementos de un programa se puede especificar con los modificadores **`public`**, **`internal`** y **`private`**.
  - Además, el nivel de acceso del **setter** de una constante, variables propiedad y subscript puede hacerse más restrictivo que el del getter usando **`private(set)`** o **`internal(set)`**.

```
private(set) var edad = 0
```

- Y puede combinarse con un nivel de acceso para el getter:

```
public private(set) var edad = 0
```

Reglas

- No pueden definirse elementos en función de otros con un nivel de acceso más restrictivo.
  - Ejemplos:
    - Una variable no puede definirse como pública y ser de un tipo interno, por que el tipo puede no estar disponible en todos los sitios donde se pueda usarse la variable.
    - Igualmente, el tipo de los parámetros y el tipo de retorno de una función no pueden tener un nivel de acceso más restrictivo que el de la función.

- El nivel de acceso de un tipo afecta al nivel de acceso de sus miembros (*propiedades, métodos, inicializadores, subscripts)*.
  - Si se define un tipo con un nivel de acceso privado:
    - sus miembros tendrán por defecto un nivel de acceso privado.
  - Si se define un tipo con un nivel de acceso interno o público:
    - sus miembros tendrán por defecto un nivel de acceso interno.
- El nivel de acceso de un tipo tupla es igual al nivel más restrictivo de los tipos usados en la tupla.

- El nivel de acceso de un tipo función se calcula como el nivel más restrictivo de los tipos de sus parámetros y su tipo de retorno.
  - Si el nivel de acceso calculado no encaja con el valor que impone el contexto, deberá especificarse explícitamente cuál es el nivel de acceso de la función.
- Los tipos de los valores raw y asociados de un enumerado no pueden un nivel de acceso más restrictivo que el enumerado.

- Tipos anidados:
  - Si el nivel de acceso de un tipo es privado, el nivel de acceso de sus tipos anidados es por defecto privado.
  - Si el nivel de acceso de un tipo es interno o público, el nivel de acceso de sus tipos anidados es por defecto interno.
- El nivel de acceso de una subclase no puede ser menos restrictivo que el de su superclase.
  - Los elementos heredados de la superclase pueden sobreescribirse para hacer su nivel de acceso menos restrictivo.
- Una constante, variable o propiedad no puede tener un nivel de acceso menos restrictivo que el de su tipo.

- Los getter y setter de las constantes, variables, propiedades y subscript tienen automáticamente el mismo nivel de acceso que el de su elemento.
  - Puede hacerse más restrictivo el nivel de acceso del setter que el del getter usando **`private(set)`** o **`internal(set)`** delante de la definición de la propiedad, variable, constante o subscript.

```
private(set) var edad = 0
```

  - Y puede combinarse con un nivel de acceso para el getter:

```
public private(set) var edad = 0
```

- Los inicializadores creados por el desarrollador pueden tener un nivel de acceso menor que el del tipo que inicializan.
  - Pero los inicializadores requeridos deben tener el mismo nivel de acceso que la clase a la que pertenecen.
- El "inicializador por defecto" tiene el mismo nivel de acceso que el del tipo que inicializa.
  - Pero si el tipo es publico, el "inicializador por defecto" es interno.
- El "inicializador memberwise por defecto" de una estructura tiene un nivel de acceso privado si alguna de las propiedades almacenadas de la estructura es privada; en caso contrario es interno.

- Al definir un protocolo puede indicarse un nivel de acceso.
  - Los elementos definidos en un protocolo tienen el mismo nivel de acceso que el protocolo.
- Si se define un nuevo protocolo que herede de uno ya existente, el nuevo protocolo no puede tener un nivel más restrictivo del que tiene el protocolo del que hereda.
- Un tipo puede ser conforme con un protocolo con un nivel de acceso más restrictivo.

- Al extender un tipo, el nivel de acceso de los elementos añadidos es por defecto igual al de los ya existentes.
- En las extensiones usadas para hacer que un tipo sea conforme con un protocolo, no puede ponerse explícitament un nivel. de acceso.
- El nivel de acceso de un tipo genérico o de una función genérica es el más restrictivo del que tienen el propio tipo o función, y de los de las restricciones de los tipos de los parámetros.
- Los tipos creados con typealias son tratados como tipos distintos en lo que se refiere al nivel de acceso.
  - El alias de un tipo debe tener un nivel de acceso igual o mas restrictivo que el del tipo en el que se basa.
- etc…

Operadores Avanzados

Operadores de Bits

- ~ es un NOT.
- & es el AND
- | es el OR
- ^ es el XOR
- << es el desplazamiento a la izquierda
- >> es el desplazamiento a la derecha

Operadores con Desbordamieinto

- Son: `&+` `&-` `&*` `&/` `&%`
  - Si se produce desbordamiento el programa no se muere.
  - La división por cero es cero.

Sobrecargar Operadores

- Las clases y estructuras pueden proporcionar una implementación de los operadores existentes.

```
struct Vector {
    var x: Int
    var y: Int
}

func + (v1: Vector, v2: Vector) -> Vector {
    return Vector(x: v1.x + v2.x, y: v1.y + v2.y)
}

var a = Vector(x:1, y:3)
var b = Vector(x:6, y:2)

a + b  // Vector(x:7, y:5)
```

- Los operadores unarios llevan **`prefix`** o **`postfix`** delante de **`func`** para indicar si son prefijos o sufijos.

```
prefix func - (v: Vector) -> Vector {
    return Vector(x: -v.x, y: -v.y)
}

var a = Vector(x:1, y:3)

-a  // Vector(x:-1, y:-3)
```

- En los operadores de asignación compuesta, el primer parámetro de la función es **inout**.

```
func += (inout v1: Vector, v2: Vector) {
    v1 = v1 + v2
}

prefix func ++ (inout v: Vector) {
    v += Vector(x:1, y:1)
}

var a = Vector(x:1, y:3)
var b = Vector(x:6, y:2)

a += b  // a es Vector(x:7, y:5)

++a     // a es Vector(x:8, y:6)
```

- Implementar los operadores de equivalencia en las clases y estructuras definidas por el usuario:

```
func == (v1: Vector, v2: Vector) -> Bool {
    return (v1.x == v2.x) && (v1.y == v2.y)
}

func != (v1: Vector, v2: Vector) -> Bool {
    return !(v1 == v2)
}
var a = Vector(x:1, y:3)
var b = Vector(x:6, y:2)
var c = Vector(x:6, y:2)

a == b   // false
a != b   // true
b == c   // true
```

Operadores Personalizados

- El desarrollador puede crearse sus propio operadores para las clases y estructuras que defina.
- Se declaran a nivel global con la palabra **operator**; se marcan con los modificadores **prefix**, **infix** o **postfix**; y para los operadores infijos se especifica entre las llaves el tipo de asociatividad (**left**, **right** o **none**) del operador y su precedencia (0..255).

```
prefix operator +++ {}

infix operator +- {associativity left precedence 140}
```

- Los nuevos operadores se implementan así:

```
prefix func +++ (inout v: Vector) {
    v += v
}

func +- (v1: Vector, v2: Vector) -> Vector {
    return Vector(x: v1.x + v2.x, y: v1.y - v2.y)
}


var a = Vector(x:2, y:3)
var b = Vector(x:6, y:2)

+++a      // a es Vector(x:4, y:6)
a +- b    // Vector(x:10, y:4)
```