



POLITÉCNICA

ETSIT
UPM

dit
UPM

Desarrollo de Apps para iOS

Introducción a Swift 4 (y 3)

IWEB 2018-2019
Santiago Pavón

ver: 2018.09.10

Introducción

Introducción

- Swift es un lenguaje moderno para crear código claro, eficiente, robusto, seguro, ...
- 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.
- Tipos predefinidos del lenguaje y definidos en el framework Foundation:
 - Int, Double, Float, Bool, String, Character, ...
 - Tuplas, Rangos, Optional, ...
 - Array, Set, Dictionary, ...
- Swift es orientado a Objetos
 - Hay Clases:
 - objetos, inicializadores, propiedades, métodos, herencia, polimorfismo, ...
 - Son tipos referencia.
 - Y además hay Estructuras y Enumerados.
 - inicializadores, propiedades, métodos, NO hay herencia, ...
 - Son tipos valor.

- Variables (*var*) y Constantes (*let*).
- Operadores (+, *, &&, ... , operadores personalizados).
- Control de flujo (*if, while, for, guard, switch, continue, ...*)
- Manejo de errores (*do try catch*).
- Propiedades (*calculadas y almacenadas*), funciones, closures, métodos. inicializadores, ...
- Protocolos, Extensiones, Genéricos,
- Control de acceso (*public, private, fileprivate, internal, ...*)
- 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** (*no mutable*)

```
let saludo: String = "hola"
```

```
let pi = 3.14
```

```
let  $\pi$  = 3.14
```

- Declarar variables con **var** (*mutable*)

```
var x = 10
```

```
var a = 1, msg = "información"
```

- No hace falta declarar el tipo si puede **inferirse** del valor asignado, o del contexto.

¿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 rápido")
}
```

- 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")
}
```

Palabras Reservadas:

Existen muchas palabras reservadas en el lenguaje que no pueden usarse como identificadores (nombres de variables, funciones, parámetros, ...).

- **class, if, var, struct, ...**

- Sin embargo, las palabras reservadas pueden usarse como identificadores poniendo acentos graves (`) antes y después del identificador.

```
var `var` = 2  
print(`var`) // 2
```

- Todas las palabras reservadas, excepto **var**, **let** y **inout**, pueden usarse como nombres de parámetros en las declaraciones y llamadas de funciones sin necesidad de usar acentos graves.

```
func a(if: Int) {}  
a(if: 3)
```

- Muchas palabras reservadas pueden usarse sin acentos graves cuando se usan para referirse a miembros de un tipo usando notación punto.

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: "")
```

Para obtener un String que represente a una instancia y poder imprimirla de forma personalizada:

- Adoptar el protocolo **CustomStringConvertible**.

```
public protocol CustomStringConvertible {  
    public var description: String { get }  
}
```

- Implementar la propiedad calculada **description**.

```
class Person: CustomStringConvertible {  
    let name: String  
    var address: String  
  
    init(name: String, address: String) {  
        self.name = name  
        self.address = address  
    }  
  
    var description: String {  
        return "Person: \(name) - Address: \(address)."  
    }  
}  
  
var p = Person(name:"Juan", address: "Madrid")  
  
print(p) // Person: Juan - Address: Madrid.
```

Tipos Numéricos

- Tipo para Enteros es **Int**
 - Otros tipos 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 para Reales:
Double // 64 bits
Float // 32 bits
 - Ejemplos:

```
let radio = 2.5 // radio es un Double
```

 - Inferencia de tipos: Double tiene prioridad

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

 - Se infiere que perimetro es 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`, `\"`, `\'`, `\xnn`, `\unnnnn`, `\Unnnnnnnnnn` (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 = ""
```

- **Swift 4** introduce literales multilínea entre tres comillas dobles **"""** y **"""**.

```
var address = """  
    Hola "Juan" y  
    Adios  
    """
```

```
// "Hola \"Juan\" y\nAdios"
```

- Las comillas finales deben estar solas en la línea y definen la indentación a ignorar de las líneas que forman el string.

- Pueden ponerse comillas en el string sin necesidad de escaparlas.

- Inicializador:

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

- Concatenar: **+** **+=**

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

- Añadir un Character al final de un String:

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

- Comparar: **==**

- En Swift 4 los caracteres son grafemas.

- Existen caracteres o grafemas que tienes varias codificaciones unicode:

- El grafema **é** puede codificarse como **0xE9** y como **0x65 0x301** (una **e** seguida de un acento).

- Dos strings son iguales si sus grafemas son iguales, independientemente de su codificación unicode.

- Propiedades:

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

- Métodos:

- **hasPrefix**, **hasSuffix**, **uppercaseString**, **lowercaseString**, **join**, **advance**, **componentsSeparatedByString**, **rangeOfString**, **description**, **splice**, **split**, ...

- **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.

- En **Swift 4** un **String** es una **Collection** y una **Sequence** de **Character**.

- Por tanto, pueden usarse las propiedades y métodos de **Collection** y **Sequence**
 - `count`, `contains`, `filter`, `isEmpty`, `reversed`, ...

```
s.count          // 7 - Contar los caracteres
s.isEmpty       // false - ¿esta vacío?
s.reversed()    // array [🚚 n ó i m a c]
```

- Iterar sobre los caracteres (grafemas) de un **String**:

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

Con **Swift 3** hay que usar la propiedad `characters` para iterar sobre los caracteres de un **String**.
o para saber cuantos caracteres contiene:

```
for c in s.characters {
    print(c)
}
s.characters.count // 7
```

- Con **Swift 4** puede accederse al contenido de un **String** usando **subscripts**.
 - El índice usado debe ser un **String.Index** o un **Range<String.Index>**.
 - No es un **Int**.

```
var s = "camión🚚"
```

```
var index = s.index(s.startIndex, offsetBy: 3)
```

```
s[index] // "i"
```

```
s[index...] // "ión🚚"
```

En swift 4 se han añadido mas cosas:

El protocolo **StringProtocol**: declara las funcionalidades de los Strings.

El tipo **Substring**: Referencia una subsecuencia de un String. Se ha creado para optimizar la representación y uso interno de los strings.

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. (Suponiendo que exista ese inicializador).

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

CGFloat

El tipo CGFloat aparecerá muchas veces cuando usemos UIKit.

Su origen está en Objective-C y se define como un float (de C).

En Swift es una estructura, y disponemos de numerosos inicializadores para realizar la conversión a otros tipos: Double, Float, ...

```
var a: CGFloat = 2.0
```

```
var b: Float = Float(a)
```

```
var c: Double = 3.5
```

```
a = CGFloat(c)
```

type(of:)

- La función **type(of:)** devuelve el tipo de la expresión pasada como argumento.

```
type(of: "22") // String.Type
```

```
type(of: 22) // Int.Type
```

```
class Demo {}
```

```
type(of: Demo()) // Demo.Type
```

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).
 - Se produce un error de ejecución.
 - 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.

- Nota: En Swift 3 se han eliminado los operadores ++ y --.

```
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.
 - Entre paréntesis, separados por comas.
 - Los valores dentro de la tupla pueden ser de distinto tipo.

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

```
// Extraer valores.
```

```
let (codigo, texto) = resultado  
print("El código 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.  
// se accede usando el nombre del componente.  
let resultado = (code: 400, msg: "Not found")  
let codigo = resultado.code  
let texto = resultado.msg  
  
// Comparar tuplas.  
// Se comparan los valores contenidos en las tuplas de  
// izquierda a derecha.  
// Los valores en las mismas posiciones en las tuplas  
// deben ser del mismo tipo, con igual número de  
// componentes, y comparables entre sí.  
//     Tipos comparables entre sí: Int, String, ...  
//     Tipos no comparables entre sí: Bool  
(10, "hola") == (12, "hola")    // false  
(10, "hola") < (10, "hola")    // false  
(10, "adios") < (10, "hola")   // true
```

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.

- Recorridos:

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

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

- En Swift 4 se introduce **One-sided Range**:
 - Es un rango en el que se proporciona solo uno de los límites.
 - El límite no especificado se determina según el contexto.

```
let numbers = [1,2,3,4,5]
```

```
numbers[...3] // [1,2,3,4]
```

```
numbers[2...] // [3,4,5]
```

```
for n in numbers[2...] {  
    print(n) // 3 4 5  
}
```

- El tipo **Range** en realidad es una estructura genérica:

```
struct Range<T> {  
    let lowerBound: T  
    let upperBound: T  
}
```

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 una copia de 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    // Se asigna una copia de a a b.
a = "tres"  // a cambia, pero b no cambia.
print(a)    // "tres"
print(b)    // "uno"
```

Type Aliases

- Crear un nombre alternativo para un tipo ya existente:

```
typealias Edad = Int  
typealias Edades = [Int] // Array de enteros
```

- Pueden usar parámetros genéricos:

```
typealias Almacen<T> = [String:T]  
  
// Almacen es un diccionario genérico con  
// un String para la clave, y los valores  
// genéricos.
```

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.
- 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 { . . . }
```

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

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

- El uso de `?` para declarar un Optional es solo 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? = Int(str)

// 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
}
```

- El inicializador **Int(*String*)** es failable, puede fallar, y por eso devuelve Int?.
- Por la inferencia de tipos, se podría poner: `let edad = Int(str)`

- 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:**

- Es una condición usada en las sentencias `if`, `while`, `repeat`, `guard` que
- comprueba si un `Optional` contiene un valor
- y si lo contiene, enlaza (inicializa) el valor extraído del `optional` con una constante o variable.

```
if let miEdad = Int(str) {  
    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`.
 - `Int(str)` es de tipo `Int`?
 - `miEdad` es de tipo `Int`
- No necesito usar `!` para extraer el valor porque `miEdad` no es un `Optional`.

- **Varios Optional Binding, cláusulas condicionales y patrones:**

- Las sentencias **if**, **while**, **guard**, ... pueden tener varias cláusulas condicionales normales, varios Optional Bindings, varios patrones case.
 - Cada Optional Binding lleva el prefijo **let** o **var**, y separados por **comas**.
 - Las cláusulas condicionales no llevan prefijos, y separados por **comas**.
 - Los patrones case llevan el prefijo **case**, y separados por **comas**.

```
var strX = "1"  
var strY = "2"  
var strZ = "0"  
var punto = (5,2)
```

```
if let x = Int(strX), x > 0,  
   let y = Int(strY),  
   let z = Int(strZ), z < x, z < y ,  
   case let (r,t) = punto {  
       print(x, y, z, r, t)  
   }
```

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 de 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 o no se devuelva un valor.
 - 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? = nil
let n = m ?? "defecto" // n es "defecto"
```

Arrays

Arrays

- Un array es una lista ordenada de valores 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:

```
[] as Array<String> // Literal array vacío.  
["Pepe", "Luis"] // Literal con varios valores.
```

- Crear un array:

```
var names1: [String] = []  
  
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](repeating: "John", count: 5)  
// 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", at:3)
           // insertar en una posición.
names.remove(at: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.

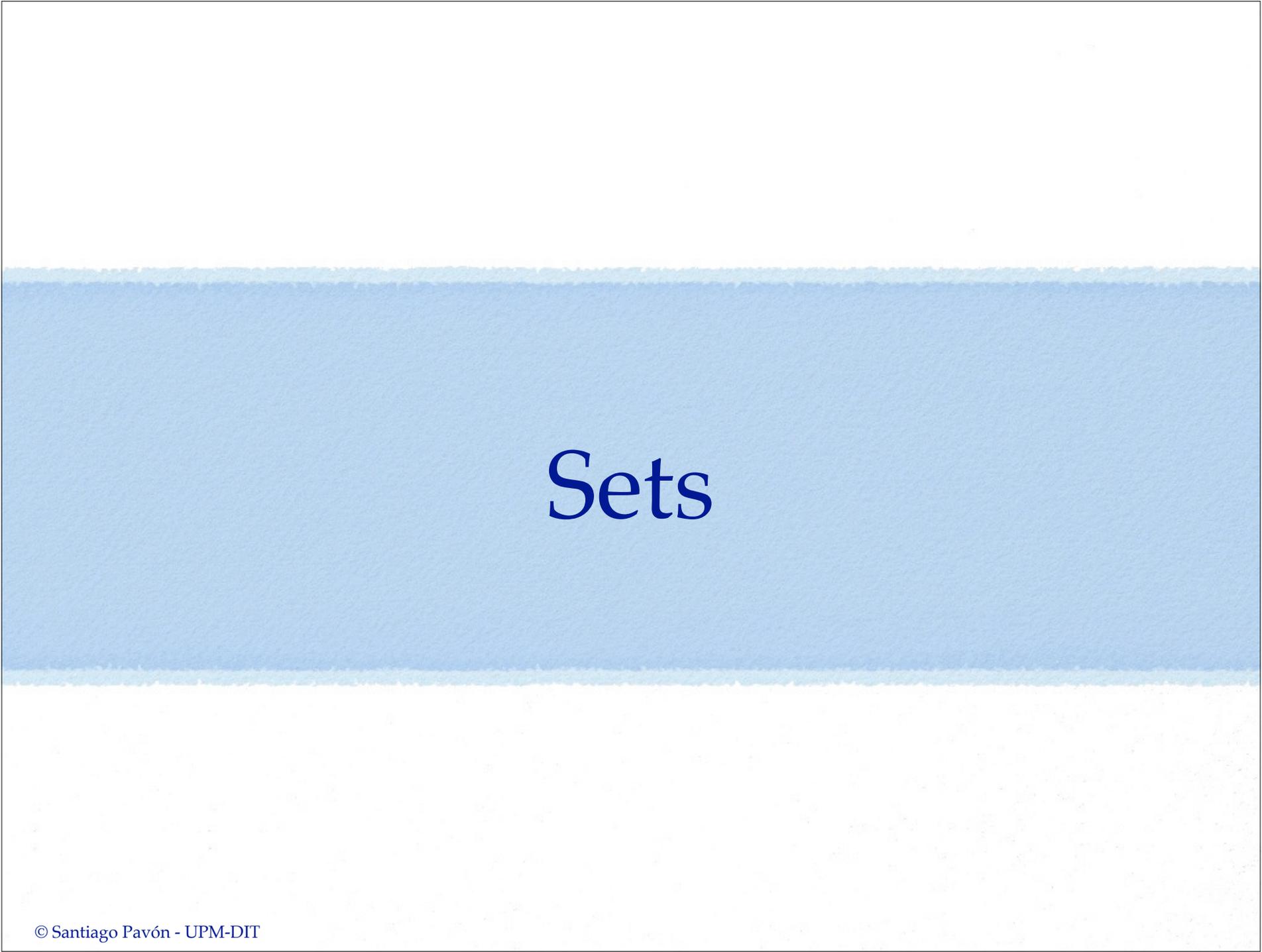
```
var z = [1,2]
z[3]   // ERROR DE EJECUCIÓN
```

- Recorrer un array:

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

```
for (index, value) in names.enumerated() {  
    print("Posición \ (index): \ (value)")  
}
```

```
// enumerated es un método de Sequence.
```



Sets

Sets

- Un **Set** es un **conjunto** de valores del mismo tipo.
 - En un conjunto los valores no están duplicados, ni están ordenados.
- 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.isSubset(of:s0) // true (subconjunto)
```

```
s2.isSubset(of:s0) // true
```

```
s1.isStrictSubset(of:s0) // false (subconjunto pero no igual)
```

```
s2.isStrictSubset(of:s0) // true
```

```
s0.isSuperset(of:s2) // true (superconjunto)
```

```
s0.isStrictSuperset(of:s2) // true (superconjunto pero no igual)
```

```
s0.isStrictSuperset(of:s3) // false
```

```
s0.isDisjoint(with:s2) // false (disjuntos)
```

```
s0.isDisjoint(with:s4) // true
```

NOTA: Estos métodos también aceptan un array como argumento:

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

- Unión, intersección, ... :

```
s0.union(s3) // Devuelve ["Uno", "Dos", "Tres", "Cuatro"]
s0.subtracting(s3) // Devuelve ["Tres"]
s0.intersection(s3) // Devuelve ["Uno", "Dos"]
s0.symmetricDifference(s3) // Devuelve ["Tres", "Cuatro"]
```

- Estos métodos devuelven una nueva instancia sin modificar el valor sobre el que se aplican

- Modificando el valor de la variable sobre la que se aplican:

```
s0.formUnion(s3) // Modifican s0.
s0.subtract(s3) // s0 debe declararse con var.
s0.formIntersection(s3)
s0.formSymmetricDifference(s3)
```

- Recorrer un Set:

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

Algunas cosas añadidas en Swift 4:

- Añadido el método `filter` para filtrar sus valores.

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:

```
[:] as Dictionary<String,String> // 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 asociado a la clave,
    // o nil (si no existe esa clave).

let n2 = codes["J", default:"Cádiz"] // Añadido en Swift 4
    // devuelve un String con el valor asociado a la clave,
    // o el valor por defecto (si no existe esa clave).

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

let v = codes.removeValue(forKey:"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 conformes con Hashable.
 - También los miembros de las enumeraciones sin valores asociados.

Algunas cosas añadidas en Swift 4:

- Inicializador **Dictionary(uniqueKeysWithValues:)** para inicializar un diccionario usando una secuencia de tuplas clave-valor.
- Inicializador **Dictionary(,uniquingKeysWith:)** para evitar sobrescrituras cuando hay claves repetidas, indicando como se resuelven estos conflictos.
- Inicializador **Dictionary(grouping:,by:)** para inicializar un diccionario usando una secuencia y agrupando los valores en arrays.
- Método **filter** para filtrar sus valores.
- Método **mapValues** para mapear los valores.
- Método **merge** para mezclar dos diccionarios.
- ...

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

- Foundation era un framework Objective-C que proporciona varios tipos, clases, protocolos, etc. de interés general.
- Originalmente era un framework Objective-C.
 - Poco a poco se ha adaptado para que su uso sea más Swift, ocultando detalles de Objective-C.
- En Objective-C los elementos proporcionados por Foundation llevan el prefijo NS en su nombre, y son principalmente clases (sus instancias se manejan por referencia) y protocolos.
- Con las diferentes versiones de Swift se han añadido nuevos tipos y pasarelas a tipos ya existentes.
 - Muchos tipos Swift no llevan el prefijo NS, y **se manejan por valor**.
 - Las instancias de estos tipos son mutables o inmutables dependiendo de como se declaren: **var** o **let**.
 - **Data, Date, DateComponents, DateInterval, Decimal, IndexPath, IndexSet, Measurement, URL, URLComponents, URLRequest, URLQueryItem, ...**
- Consultar la documentación de Foundation y de la Librería Estandar de Swift para ver los detalles de los tipos disponibles.

Search

- Documentation and API Reference ⌘0
- Xcode Help
 - What's New in Xcode
 - Release Notes
- Quick Help for Selected Item ⌘?
- Search Documentation for Selected Text ⌘⌘/

Swift

App Frameworks

- AppKit
- Foundation
- Objective-C
- Swift Standard Library
- UIKit
- WatchKit

App Services

- Accounts
- AddressBook
- AddressBookUI
- AdSupport
- ApplicationServices
- CallKit
- ClockKit
- CloudKit
- Contacts
- ContactsUI
- Core Data
- Core Foundation
- Core Location
- Core Motion
- Core Spotlight
- Core Text
- EventKit
- EventKitUI
- HealthKit
- HealthKitUI

Framework

Foundation

Access the essential classes that define basic object behavior, data types, collections, and operating-system services. Incorporate design patterns and mechanisms that make your apps more efficient and robust.

Language
Swift | Objective-C

SDKs
iOS 2.0+
macOS 10.0+
tvOS 9.0+
watchOS 2.0+

Overview

The Foundation framework defines a base layer of Objective-C classes. In addition to providing a set of useful primitive object classes, it introduces several paradigms that define functionality not covered by the Objective-C language. The Foundation framework is designed with these goals in mind:

On This Page
[Overview](#)
[Symbols](#)

- Provide a small set of basic utility classes.

- **NSMutableDictionary, NSMutableArray**
- **Date, Calendar, DateFormatter, DateComponents, NSDate, NSCalendar, NSDateFormatter, NSDateComponents**
 - Fechas y horas.
- **NSNumberFormatter, NSNumberFormatter**
 - Formatear números.
- **NSRegularExpression**
 - Expresiones regulares.
- **Operation, OperationQueue, 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.
- **UserDefaults, UserDefaults**
 - Preferencias de usuario.
- **NSProgress**
 - Informes de progreso.
- **NSEnumerator**
 - Recorrer los elementos contenidos en una colección.
- ...

Control de Flujo

Condicional: if

- 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")  
}
```

- También pueden llevar varias cláusulas condicionales, optional bindings, patrones case, type casting, patrón comodín "_", ...

```
var strX = "1"  
var strY = "2"  
var strZ = "0"  
var punto = (5,2)
```

```
if let x = Int(strX), x > 0,  
   let y = Int(strY),  
   let z = Int(strZ), z < x, z < y ,  
   case let (r,t) = punto {  
       print(x, y, z, r, t)  
   }
```

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".characters { // Swift 3. En Swift 4 sin .characters  
    print(c)  
}
```

```
for v in array {  
    print(v)  
}
```

- Iterar sobre las claves y valores de un diccionario usando una tupla:

- Se usa un tupla para representar los datos (*Tuple Pattern*).

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

- El bucle for-in puede llevar varias cláusulas condicionales con **where**, type casting, patrón comodín "_", ...

```
for _ in 1...10 {  
    print("*") // *****  
}
```

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

Rangos

- Rango incluyendo valores izquierdo y derecho (1 ...10)

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

- Solo en Swift 4: One-sided Rango: el contexto determina el valor del límite no especificado

```
let numbers = [1,2,3,4,5]  
for n in numbers[2...] {  
    print(n)      // 3 4 5  
}
```

Bucle For: inicialización-condición-incremento

- Este tipo bucle **for** se ha eliminado en Swift 3.

```
for var i = 0 ; i < 4 ; i += 1 {  
    print(i)  
}
```

- El comportamiento de este tipo de bucle se puede obtener usando rangos:

```
for i in 0..<4 {  
    print(i)    // 0 1 2 3  
}
```

- Para hacer un recorrido en orden inverso se puede usar el método **reversed**:

```
for i in (0..<4).reversed() {  
    print(i)    // 3 2 1 0  
}
```

- Para incrementos distintos de 1, se puede usar el protocolo **Strideable**, que lo implementan muchos de los tipos numéricos.

```
for i in stride(from: 0, to:4 , by: 0.5) {  
    print(i)      // 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5  
}  
for i in stride(from: 0, through: 4, by: 0.5) {  
    print(i)      // 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0  
}
```

- La función **stride** devuelve una instancia de **StrideTo** o **StrideThrough**, que son conformes con **Sequence**, y por tanto se puede hacer todo lo que soportan las secuencias (**map**, **filter**, **forEach**, ...):

```
stride(from: 0, to:4 , by: 0.5).forEach({print($0)})  
// 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5
```

Bucle: while

- Iterar hasta que la condición es false.

```
var n = 10
```

```
while n > 0 {  
    n -= 1  
}
```

- Pueden llevar varias cláusulas condicionales, optional bindings, patrones case, type casting.

```
while let n = next(), n > 0 {  
    print(n)  
}
```

Bucle: repeat-while

- Iterar mientras que la condición es true.

```
var n = 10
```

```
repeat {  
    n -= 1  
} while n > 0
```

switch

- Compara un valor con varios patrones para ver cual encaja:

```
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 los case pueden usarse rangos:

```
let age = 22

switch age {
case 0:
    print("recien nacido")
case 1...12:
    print("Niño")
case 13...18:
    print("Adolescente")
case ...65: // Recordad que One-Sided Ranges se añadieron en Swift 4.
    print("Adulto")
default:
    print("Viejo")
}
```

- En Swift 4 las expresiones no tienen que ser Int.

```
let vote = 0.2

switch vote {
case ..<0.45:
    print("👎")
case ..<0.9:
    print("👍")
default:
    print("👏👏👏👏")
}
```

- Usar tuplas para comprobar varios valores (*Tuple Pattern*).

```
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 patrón comodín "_" encaja con cualquier valor, pero el valor con el que encaja se ignora.

- **Value bindings:** Enlazar los valores que encajan en el case 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 cláusulas **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")  
}
```

- Usar los patrones Type-Casting **is** y **as**.
 - Es un switch no pueden usarse **as?** ni **as!**.

```
let a:Animal = Dog()

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")
}
```

- Ejemplo: Patrones Tuple, Type-Casting y comodín juntos:

```
let people: [(String, Any)] = [ ("Pepe", 44),  
                                ("Ana", "Toledo"),  
                                ("Juan", true)]  
  
for person in people {  
  
    switch person {  
    case let (name, is Int):  
        print("Nombre \(name) y edad")  
    case (let name, let address as String):  
        print("Nombre \(name) y dirección \(address)")  
    case let (name, _):  
        print("Otra cosa")  
    }  
}
```

- Usar el **patrón optional** en los casos.

- Este patrón solo encaja con valores opcionales que contengan un valor (que no sean nil).

```
let people: [(String, String?)] = [("Pepe", nil),  
                                   ("Ana", "Toledo"),  
                                   ("Juan", "Lugo")]  
  
for person in people {  
  
    switch person {  
    case let (name, address?):  
        print("Nombre \(name) y dirección \(address)")  
    case let (name, _):  
        print("Nombre \(name)")  
    }  
}
```

- Usar el **patrón Enumeration Case** en los cases.
 - Este patrón solo encaja con los valores de un enumerado.

```
enum Season {  
    case spring, summer, autumn, winter  
}
```

```
let s: Season = .summer
```

```
switch s {  
case .spring:  
    print("Estamos en primavera.")  
case .summer:  
    print("Estamos en verano.")  
case .autumn:  
    print("Estamos en otoño.")  
case .winter:  
    print("Estamos en invierno.")  
}
```

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.
- **guard** siempre 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(), ...
- **guard** puede llevar varias cláusulas condicionales, optional bindings, patrones case, type casting, patrón comodín "_", ...

- 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 poniendo varias condiciones en una sola sentencia guard:

```
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  
    }  
}
```

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 switch, 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 row in tablero {  
  for column in row{  
    if tablero[x][y] == 5 {  
      print("Encontrado")  
      break fuera  
    } else {  
      print("Sigo buscando")  
    }  
  }  
}  
print("Estoy fuera")
```

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 tiene un nombre de parámetro y una etiqueta de argumento.
 - 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 se ponen solo los paréntesis ().
- Si no se devuelve nada, se omite la flecha y el tipo de retorno.
 - En realidad se devuelve una tupla vacía del tipo **Void**.

```
func dump() {  
    print("datos")  
}  
dump() // datos
```

- Invocar la función:

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

- **Nota:** Si no nos interesa el resultado devuelto, asignarlo a "_" para que Xcode no muestre warnings.

```
let _ = wellcome(name: "Pedro")
```

- Otra forma de evitar el warning es poner la directiva **@discardableResult** en la definición la función.

- 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(city: "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(city: "Madrid") {
    print("Posición= \(pos.latitude) , \(pos.longitude)")
} else {
    print("No encontrada")
}
```

Etiquetas de los Argumentos

- Los parámetros de las funciones tienen:
 - Un nombre de parámetro.
 - Se usa en el cuerpo de la función.
 - Los nombres de los parámetros deben ser únicos.
 - Una etiqueta de argumento
 - En las llamadas a la función, se pone delante del valor del argumento.
 - Puede repetirse las etiquetas de los argumentos.
 - Por defecto, si no se proporciona una etiqueta de argumento, se usa el nombre del parámetro.

- Ejemplo:

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

- **p1** y **p2** son los nombres de los parámetros, y también las etiquetas de los argumentos.

- Si no nos gusta el nombre del parámetro como etiqueta de argumento, podemos indicar que etiqueta de argumento queremos usar escribiéndola delante del nombre del parámetro.
- Si no se quiere usar la etiqueta de argumento en un parámetro, usar "_" como etiqueta de argumento.

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

Valor por defecto de parámetro

- Al definir una función, puede definirse un valor por defecto para los parámetros.

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

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

- Situar los parámetros con valores por defecto al final de la lista de parámetros para evitar confusiones y ambigüedades en las llamadas a las funciones entre los parámetros normales y los que tienen valores por defecto.
- En las llamadas a la función, los parámetros opcionales que se usen deben proporcionarse en el mismo orden que en la declaración de la función.

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(numbers: 22, 34, 11) // 67
```

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

Parámetros: Son Constantes

- 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
}
```

- Pero se pueden crear variables nuevas que si se pueden modificar:

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

Parámetros In-Out

- Para que los cambios realizados en un parámetro dentro del cuerpo de una función persistan al finalizar la ejecución de una función.
 - Este comportamiento se llama **copy-in copy-out**, y también **call by value result**.
- En la declaración de la función se escribe **inout** delante del tipo del parámetro.
- En la llamada a la función se añade **&** delante del argumento.
 - Solo pueden pasarse variables en los parámetros inout.

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

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

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

func increment(_ value: inout 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
```

Los parámetros con valor por defecto no pueden ser inout.

Los parámetros variadic no pueden ser inout.

Copy-in copy-out:

- Cuando se llama a la función: se copia el valor de la variable usada en el argumento inout y se pasa ese valor al parámetro.
- Cuando termina la función: se copia el valor del parámetro en la variable que se paso como argumento.
- Por tanto, no debe accederse a la variable pasada como argumento inout desde dentro de la función, ya que al finalizar la función se sobrescribirá el valor de la variable.
- No pasar la misma variable en varios parámetros inout. Al terminar la función se sobrescribiría el valor de la variable varias veces, y no está definido cuál sería el resultado final.
- La copia out solo se realiza al terminar la función.
 - Si en la función se creo un closure o una función anidada, no se actualizará el valor de la variable inout aunque se cambie el valor del parámetro en un futuro.

Tipo Función

- Cada función tiene un tipo.
 - Formado por el tipo de los parámetros y el tipo de retorno.
 - Las etiquetas de los argumentos no son parte del tipo, son parte del nombre de la función.
- 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 `() -> ()` o `() -> Void`
 - Tipo función que no toma parámetros y no devuelve nada.
- Si el tipo de una función incluye mas de una flecha `->`, los tipos de las funciones se agrupan de derecha a izquierda.
 - `(Int) -> (Int) -> Int` significa `(Int) -> ((Int) -> Int)`.
- El tipo de las funciones **throwing** o **rethrowing**, incluye la palabra **throws** o **rethrows** antes de la flecha.
 - Por ejemplo: `(Int) throws -> Int`

- 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 guardada 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(v1:10, v2:20, operation:sum)  
// Paso una función en último argumento
```

```

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, condition: positive) // Imprime 2, 44 y 9

```

- Las etiquetas de los argumentos no son parte del tipo, son parte del nombre de la función.

```
func sum(_ a: Int, _ b: Int) -> Int { // (Int,Int) -> Int
    return a + b
}
func sum(a: Int, b: Int) -> Int { // (Int,Int) -> Int
    return 5*a + b
}
func sum(a: Int, f: Float) -> Int { // (Int,Float) -> Int
    return a + Int(f)
}
```

- Las etiquetas de los argumentos deben usarse en la llamadas a la función.

```
sum(1, 2) // 3
sum(a:1, b:2) // 7
sum(a:1, f:2.5) // 3.5
```

- El uso de las etiquetas de argumentos puede usarse para eliminar ambigüedades.

```
let ope0 = sum // Error: uso ambiguo de sum

let ope1 = sum(_:_) // tipo inferido de ope1 es (Int,Int) -> Int
let ope2 = sum(a:b:) // tipo inferido de ope2 es (Int,Int) -> Int
let ope3 = sum(a:f:) // tipo inferido de ope3 es (Int,Float) -> Int
```

- Aquí no se usan las etiquetas de los argumentos en la llamadas a las funciones referenciadas.

```
ope1(1,2) // 3
ope2(1,2) // 7
ope3(1,2) // 3.5
```

- Al escribir explícitamente un tipo función no se usan las etiquetas de los argumentos,
 - pero pueden usarse como documentación usándolas como si fueran nombres de parámetros y precedidas de "_".

```
func multiply(_ a: Int, by b: Int) -> Int {  
    return a * b  
}
```

```
let ope: (_ a:Int, _ by:Int) -> Int
```

```
ope = multiply  
ope(3,2) // 6
```

Funciones que no retornan nunca

- Las funciones que nunca retornan devuelven el tipo **Never**.

```
func fatalError(msg: String) -> Never { ... }
```

```
func operation<T>(b: T -> Never) { ... }
```

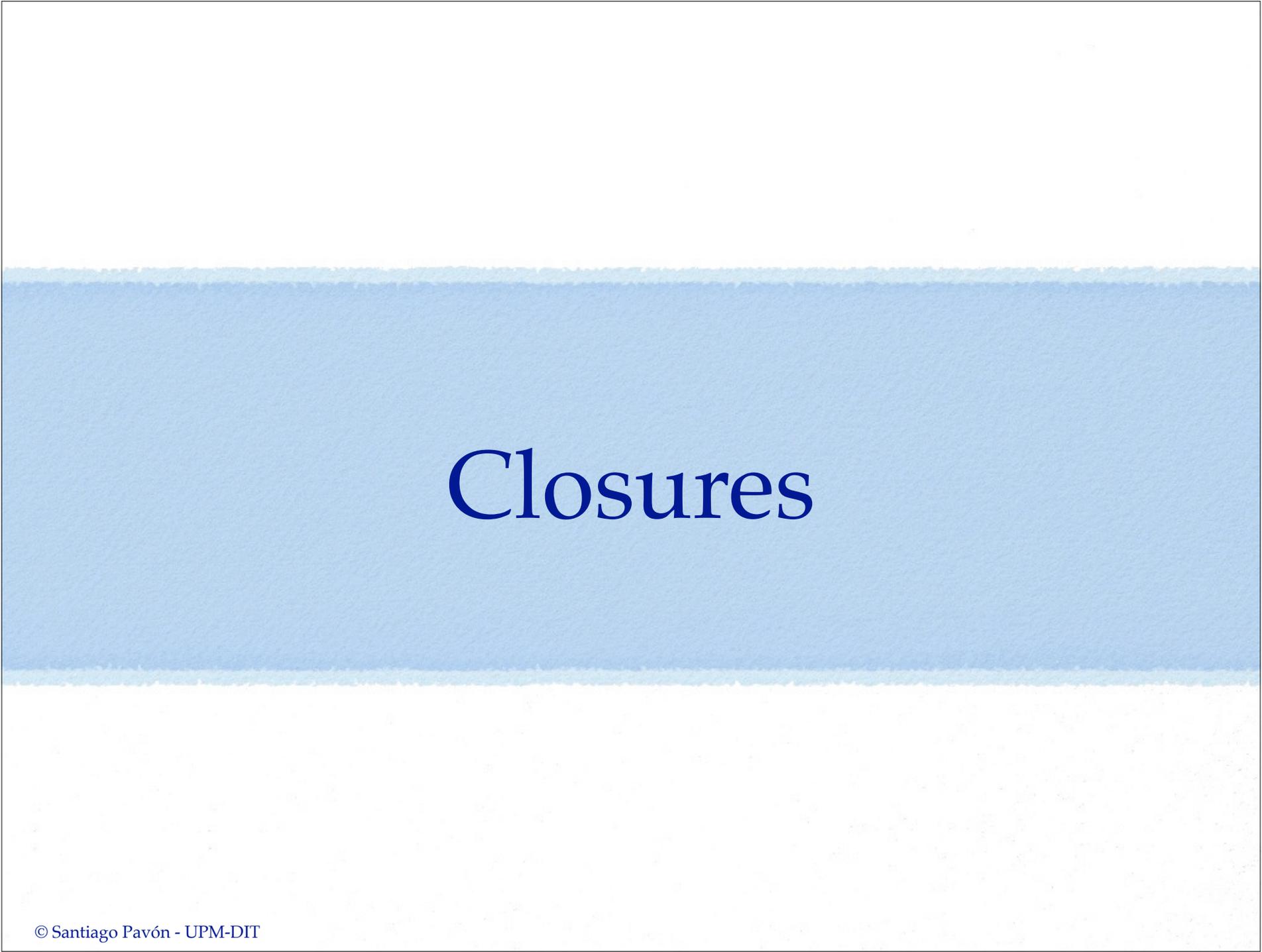
- Nota: Antes de Swift 3 se marcaban con el atributo `@noreturn`

Conversiones entre Tipos Función

- Desde Swift 2.1 se soportan conversiones entre distintos tipos función.
 - Se soporta **covarianza** en el tipo devuelto por las funciones.
 - Se soporta **contravarianza** en los tipos de los parámetros.
- Esto significa que podemos pasar un valor del tipo **función Any -> Int** a un sitio que espere un tipo **función String -> Any**.

```
func v(a:Any) -> Int { return 25 }  
  
let f: (String) -> Any = v  
  
let r = f("hola")
```

- El tipo del parámetro de la función asignada (**Any**) debe ser menos específico que el tipo esperado (**String**).
- El tipo de retorno de la función asignada (**Int**) debe ser más específico que el tipo de retorno esperado (**Any**).
- Evidentemente, esto aplica también a las closures.



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.
- Las closures 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**.
 - Se asignan o pasan copias de las referencias.

Expresión Closure

- La sintaxis de una expresión closure es:

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

- 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 ordenado(_ values: [Int], condition: (Int,Int) -> Bool) -> [Int] {
    var values = values
    for _ in 0..
```

- 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 el tipo de los parámetros y el tipo de retorno:

- Las expresiones closure se pueden simplificar si por inferencia de tipos sabemos cuál es el tipo de los parámetros y el tipo de retorno.
- Se han eliminado los paréntesis, el tipo de los parámetros, la flecha y el tipo de retorno, ya que por inferencia de tipos se conocen los tipos.

```
let t1 = ordenado(table,  
                  condition: {(v1: Int, v2: Int) -> Bool in  
                              return v1 > v2})
```

```
let t2 = ordenado(table,  
                  condition: {v1, v2 in return v1 > v2})
```

- 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 = ordenado(table,  
                  condition: {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 = ordenado(table,  
                  condition: {$0 > $1})
```

- Usar el nombre del 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 = ordenado(table,  
                  condition: >)
```

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

```
ordenado(table) {v1, v2 in v1 > v2}  
ordenado(table) {$0 > $1}
```

- Nota: Si el único parámetro de la función es la closure, pueden también 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/función 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) -> Int {  
  
    var total = 0 // variable local de cada llamada a la función  
  
    return { (v:Int) -> Int in // Este closure captura la variable  
        total += v           // total definida en el ámbito de  
        return total        // cada llamada a esta función.  
    }  
}  
  
let ac1 = makeAcumulator() // ac es del tipo (Int) -> Int  
let ac2 = makeAcumulator() // ac es del tipo (Int) -> Int  
  
ac1(1) // total del ámbito de ac1 es 1  
ac2(2) // total del ámbito de ac2 es 2  
ac1(4) // total del ámbito de ac1 es 5  
ac1(3) // total del ámbito de ac1 es 8  
ac2(5) // total del ámbito de ac2 es 7  
ac2(5) // total del ámbito de ac2 es 12
```

Cuidado con no crear
bucles de retenciones

Escape Closures

- Se dice que un parámetro de tipo closure escapa (*escape*) a una función cuando:
 - el parámetro de tipo closure de la función se llama después de que la función haya terminado.
 - Ejemplo típico: uso de callbacks en llamadas asíncronas.
- Por defecto, los parámetros closure no escapan.
- Si un parámetro closure puede escapar, debe marcarse con el atributo **@escaping**.

```
var table = [() -> Void]() // Almacén de closures

func save(closure: @escaping () -> Void) { // Guarda en la tabla una closure pero
    table.append(closure) // no la invoca/ejecuta.
}

var n = 5

save(closure: {n += 1}) // Guarda en la tabla una closure
save(closure: {n *= 2}) // Guarda en la tabla otra closure

for f in table { // Ahora ejecuto todas las closures guardadas
    f()
}
n // 12
```

- Dependiendo de que el parámetro closure pueda escapar o no, el compilador realiza distintas optimizaciones.

- Los parámetros de tipo closure marcados con **@escaping** deben referirse a las propiedades y métodos de **self** usando explícitamente "**self.**".

```
func apply1(closure:() -> Void) {
    closure()           // la closure no escapa a la función.
}

func apply2(closure: @escaping () -> Void) {
    operacionAsincrona(closure) // la closure escapa a la función.
}

class Demo {
    var n = 5

    func demo() {
        apply1(closure: {n += 1}) // el uso de self. es implícito.
        apply2(closure: {self.n += 1}) // requerido uso explícito.
    }
}
```

Autoclosures

- Se usa para retrasar la evaluación de una expresión pasada como argumento a una función.
 - Se hace envolviendo automáticamente la expresión en una closure sin argumentos.
- El parámetro de la función es de tipo closure y se marca con el atributo **@autoclosure**.
 - la expresión no se ejecuta hasta que se llame a la closure autocreada, no al llamar a la función.
 - y la closure devuelve el valor de la expresión que envuelve.

```
var n = [2, 3, 4, 5]
```

```
func get(_ closure: @autoclosure () -> Int, if condition: Bool) -> Int {  
    return condition ? closure() : 0  
}
```

```
get(n[0],          if: true) // Devuelve 2.  
get(n.remove(at:0), if: false) // Devuelve 0. No ejecuta la expresion.  
get(n.remove(at:0), if: true) // Ejecuta la expresión que modifica n  
                                // borrando el primer valor. Devuelve 2.  
get(n[0],          if: true) // Devuelve 3.
```

- Si un parámetro autoclosure puede escapar, se marca con **@autoclosure @escaping**.

Estructuras

Estructuras

- Son tipos manejados por valor.
 - Las instancias se asignan o pasan por valor (*una copia del valor*).
- Tienen propiedades, métodos e inicializadores.
- Diferencias con las clases:
 - Las clases son tipos referencia y las estructuras son tipo valor.
 - Las estructuras no tienen métodos De inicializadores.
 - Las estructuras no soportan herencia.
 - No aplican los conceptos 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 (llamado *memberwise initializer*) 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 una copia de 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.
// c1 y c2 son dos instancias separadas.
```

```

var c = Color(red: 50, green: 50, blue: 50)
var 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 la copia hasta que no es absolutamente necesario.
 - No hay que preocuparse por la eficiencia.

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*) con 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 una copia de la referencia a la instancia.

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

let f2 = f1

// f1 y f2 apuntan/referencian al mismo objeto

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

f1.name = "Cuadrado"

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

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 . . .

Enumerations

Enumeration

- Una enumeración define un nuevo tipo formado por un grupo de valores excluyentes entre sí.
 - El valor de datos 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.
- 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

- Los valores miembros de un enumerado pueden tener valores asociados.
- El tipo de los valores asociados de cada 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.

```

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 += 1
        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).")
}

if case .other(let descr) = v4 { // Extraer valor asociado con if case
    print("Es un \(descr).")
}

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

```

Raw Values

- Al definir un enumerado se puede asignar un valor raw para los miembros definidos
 - Todos los valores raw de 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           // valor raw es 0  
    case tuesday         // valor raw es 1  
    case wednesday       // valor raw es 2  
    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"  
}
```

- La propiedad **rawValue** devuelve el valor raw de un miembro:

```
Season.autumn.rawValue // "o"  
Day.friday.rawValue   // 51
```

- El inicializador **init?(rawValue:)** crea una instancia de la enumeración asociada al valor raw dado.
 - Nótese que es un inicializador failable,
 - ya que puede que no exista un miembro para el valor raw dado.

```
if let s = Season(rawValue: "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 una reserva infinita de memoria para poder almacenar los valores enumerados.
- Se evita introduciendo un nivel de indirección.
- Añadiendo **indirect** delante de los casos donde exista recursividad, o delante de enum para que afecte a todos los casos.

```
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 constante, 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 la 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 lo vemos 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 .pi * radius * radius
        }
        set(newArea) {  // setter
            radius = sqrt(newArea / .pi)
        }
    }
}

var c = Circle()
c.radius           // 1.0
c.area             // 3.14159
c.area = 12.566371
c.radius           // 2.0
```

- Puede omitirse el parámetro 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 .pi * radius * radius  
        }  
        set {              // setter  
            radius = sqrt(newValue / .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 .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.
- Si una propiedad con observadores se pasa como un parámetro inout en una función, entonces los observadores se ejecutan siempre.

```
struct Circle {  
    var radius: Double = 1.0 {  
        didSet {  
            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**.

```
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
```

- En las clases, para permitir que las propiedades de tipo y calculadas puedan ser sobrescritas en subclases derivadas, hay que usar la palabra **class**, en vez de **static**.

```
class Message {  
    var header = "Hola"  
    var body = "¿Qué hora es?"  
  
    class var footer: String {  
        return "(c) IWEB"  
    }  
}
```

```
class Message2: Message {  
    override class var footer: String {  
        return "(c) IWEB-2"  
    }  
}
```

```
Message.footer // (c) IWEB  
Message2.footer // (c) IWEB-2
```

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 += 1
    }
    func up(steps:Int) {
        y += steps
    }
    func gotoOrigin() {
        x = 0; y = 0
    }
}

let p = Position() // 0 0
p.up()             // 0 1
p.up(steps: 4)    // 0 5
p.gotoOrigin()    // 0 0
```

Etiquetas de los Argumentos

- Igual que en las funciones:
 - Los parámetros de los métodos tienen un nombre de parámetro y una etiqueta de argumento.
 - Por defecto, la etiqueta de argumento es igual al nombre del parámetro, pero puede cambiarse escribiendo otra etiqueta de argumento delante del nombre del parámetro.
 - Si no se quiere usar la etiqueta de argumento en un parámetro, usar "_" como etiqueta de argumento.

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

    // . . .

    // Nombres de parametro: factor, h, v.
    // Etiquetas de argumento: factor, inHorizontal, inVertical
    func scale(factor:Int, inHorizontal h:Bool, inVertical v:Bool) {
        if h { x *= factor }
        if v { y *= factor }
    }
}

let p = Position()

p.x = 3
p.y = 4

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

self

- **self** es una propiedad implícita existente en todas las instancias.
 - Apunta a la propia instancia.
- Se usa en los métodos de instancia para referirse a la propia instancia.
- Necesidad de usarlo:
 - 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.
 - Es obligatorio en el cuerpo de los parámetros que son escaping closures.
 - ...

```
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.moveToX(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 toda la instancia por uno nuevo.

```

struct Size {
    var width = 0
    var height = 0

    mutating func scale(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.scale(factor:2) // width=20 height=40
s1.zero() // width=0 height=0

let s2 = Size(width: 10, height: 20)
s2.scale(factor: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.
- 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

```

- En las clases, para permitir que los métodos de tipo puedan ser sobrescritos en subclases derivadas, hay que usar la palabra **class**, en vez de **static**.

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

    class func footer() {
        print("(c) IWEB")
    }
}

class Message2: Message {
    override class func footer() {
        print("(c) IWEB-2")
    }
}
```

```
Message.footer() // (c) IWEB
Message2.footer() // (c) IWEB-2
```

Subscript

Subscript

- Es una sintaxis usando `[y]` para acceder a los datos guardados en las enumeraciones, estructuras o clases.
 - Normalmente se usa 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 índice  
    }  
}
```

- 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.
- La forma de usar las etiquetas de argumentos y nombres de parámetros es igual que en las funciones.

```

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

    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]
    }
}

```

1			3
			1
	2		

```

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 subclasses 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 sobrescribir lo que desee modificar.

Sobrescribir

- Una clase hija puede sobrescribir 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 sobrescribiendo.
 - No se puede anteponer la palabra **override** delante de un elemento que no se esté sobrescribiendo.
 - 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 sobrescribirse 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
```

- **Sobrescribir propiedades:**

- Se pueden sobrescribir 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 sobrescribiendo).
- **Restricción:** Una propiedad readwrite no puede sobrescribirse como una propiedad read-only.
 - Si se sobrescribe 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(meters: Double) {
        distanceInMeters = meters
    }
    init(kilometers: Double) {
        distanceInMeters = kilometers * 1000
    }
}

let d1 = Distance()           // 0
let d2 = Distance(meters:5)  // 5
let d3 = Distance(kilometers: 3) // 3000
```

- Etiquetas de los argumentos (*igual que con las funciones*):
 - Los parámetros de los inicializadores tienen un nombre de parámetro y una etiqueta de argumento.
 - Por defecto, la etiqueta de argumento es igual al nombre del parámetro, pero puede cambiarse escribiendo otra etiqueta de argumento delante del nombre del parámetro.
 - Si no se quiere usar la etiqueta de argumento en un parámetro, usar "_" como etiqueta de argumento.
- 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 llamar 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:.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 Sobrescritura

- Por defecto, una subclase no hereda los inicializadores de la superclase.
 - Solo se heredan los inicializadores de la superclase 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 sobrescribe 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
}

```

- **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.

- **¿Cuándo 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 sentencias de inicialización.
- Un inicializador failable también puede delegar en un inicializador no failable.
- Un inicializador no failable nunca puede delegar en un inicializador failable.

- **Sobrescribir un inicializador failable:**

- Una subclase puede sobrescribir 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 manejadas por referencia.
- 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, car: Car) {
    self.name = name
    self.car = car
    car.owner = self
  }

  deinit {
    print("Desinicializando Persona")
  }
}

```

```

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

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

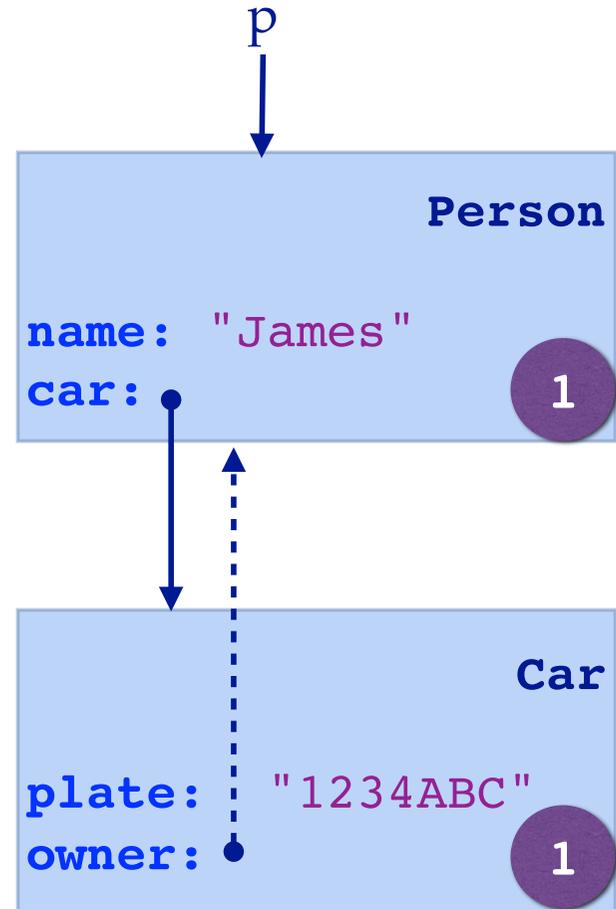
  deinit {
    print("Desinicializando Coche")
  }
}

```

```

var p = Person(name:"James", car: Car(plate: "1234ABC"))
p = Person(name:"Mike", car: Car(plate: "5678DEF"))

```



Bucles Strong con Closures

- Preliminares:
 - Las closures son Tipos Referencia.
 - y por tanto pueden ser retenidas.
 - Una closure también puede retener (capturar) instancias de una clase
 - si en el cuerpo de la closure se accede a una propiedad de la instancia, o si llama a un método de la instancia, o si retiene algo que este reteniendo la instancia, o ...
- Por tanto, también es posible crear bucles de retenciones strong al usar closures.
 - Ejemplo:
 - Una instancia "i" guarda en su propiedad "p" una closure "c".
 - es decir i.p está reteniendo 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

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

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

    deinit {
        print("Desinicializando")
    }
}

do {
    var t = MultiplicationTable(7)
    t.multiplyBy(3) // 21
}

```

La tabla de multiplicar del 7:

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

Y esta 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

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

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

    deinit {
        print("Desinicializando")
    }
}

do {
    var t = MultiplicationTable(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 sobrevivirá a la otra.

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

Manejo de Errores

Manejo de Errores

- Mientras se ejecuta un programa pueden producirse situaciones de error que hay que manejar y de las que hay que recuperarse.
 - Veremos como se capturan los errores que se producen, como crear nuestros propios errores personalizados, como lanzarlos, como se propagan, ...
- Pero si en nuestro programa no nos interesa conocer los detalles de los errores que se producen, podemos ignorarlos usando tipos Optional:
 - Así, si algo va mal, se genera un nil, e ignoramos los detalles del error.
- Aunque debemos ser conscientes de que algunas situaciones de error pueden ser irrecuperables, y lo mejor sería hacer que el programa se 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 Error

- Los tipos usados para representar errores deben ser conformes con el protocolo **Error**.
- Los tipos de error se suelen implementar típicamente como enumerados conformes con el protocolo Error,
 - con varios cases para los posibles errores,
 - y con valores asociados en los cases donde se quiera tener información adicional sobre el error.

```
enum UserError: Error {  
    case unknown  
    case noAuthorized(code: Int)  
}
```

Mas protocolos:

- LocalizedError - permiten proporcionar descripciones localizadas de los errores.

```
extension UserError: LocalizedError {
  var errorDescription: String? {
    switch self {
    case .unknown:
      return NSLocalizedString("UserErrorUnknown", comment: "Desconocido")
    case .noAuthorized:
      return NSLocalizedString("UserErrorNoAuthorized", comment: "No autorizado")
    }
  }
}
```

- RecoverableError
- CustomNSError

Lanzar Errores: **throw**

- Los errores se lanzan con **throw**
throw **UserError.noAuthorized**(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:
 - Los errores que se producen dentro de una función, pueden propagarse hacia el punto que invocó la función.
 - Capturar y manejar los errores con sentencias do-catch.
 - Capturarlos y convertirlos en valores Optionals.
 - Indicando que aunque teóricamente es posible que se produzca un error en un sitio, 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 login() throws -> String {  
    . . .  
    throw UserError.unknown  
    . . .  
}
```

- 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 comprueba 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.

```
enum UserError: Error {  
    case unknown  
    case noAuthorized(code: Int)  
}  
  
func login() throws {  
    throw UserError.noAuthorized(code: 4)  
}  
  
do {  
    try login()  
} catch UserError.unknown {  
    print("Usuario desconocido")  
} catch UserError.noAuthorized(let code) where code > 0 {  
    print("Fallo autorización positivo", code)  
} catch let error as UserError {  
    print("Error", error)  
} catch {  
    print("Error Desconocido")  
}
```

Funciones Rethrowing

- Una función o método puede declararse con **rethrows** para indicar que lanzará un error solo si uno de sus **parámetros función** lanza un error.
 - Alguno de los parámetros debe ser de tipo función.
 - Las funciones o métodos rethrowing solo pueden lanzar un error si lo ha hecho uno de los parámetros función.
 - El error lanzado por una función rethrowing puede ser:
 - el mismo que lanzó uno de sus parámetros,
 - u otro creado por nosotros.
 - Usaremos un **do-catch** para capturar el error lanzado por el parámetro, y lanzar el nuevo error desde el **catch**.
 - (En las funciones rethrowing solo está permitido lanzar errores desde el catch)
- Además:
 - Un método throwing no puede sobrescribir un método rethrowing, ni implementar un método rethrowing requerido por un protocolo.
 - Un método rethrowing si puede sobrescribir un método throwing, y puede implementar un método throwing requerido por un protocolo.

```

enum MiError: Error {
    case error1, error2, error3
}

// siempre genera un error
func e1() throws {
    throw MiError.error1
}

func demo(cb: () throws -> ()) rethrows {
    do {
        try cb()
    } catch {
        throw MiError.error3
    }
}

try? demo(cb: e1)

do {
    try demo(cb: e1)
} catch {
    . . .
}

```

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 transforma 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 ningún error, entonces podemos llamarla con **try!**.
 - Esto deshabilita la propagación de errores.
 - Pero si se lanza un error, el programa se 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 pueden usarse los patrones **is** y **as**.
 - Pero no pueden usarse **as?** ni **as!**.

```
let a:Animal = Dog()

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?>(repeating: nil, count: 64)
    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 sobrescribir 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 disponibles 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 times(task: () -> ()) {
    if self < 0 { return }
    for _ in 0..

```

```

4.odd // false
let n = Int(base:5, exponent:2) // 500
3.times {print("hola")} // hola hola hola
var x = Int(base: 3, exponent: 2) // 300
x.double() // 600
12345[3] // 2

```

```
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,
 - (Nota: Desde Swift 2 se pueden proporcionar implementaciones por defecto)
 - Solo describe cómo debe ser la implementación.
- Los protocolos son **adoptados** por clases, estructuras y enumeraciones,
 - e 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 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 de acceso requerido.
 - Si la propiedad es de tipo, usar siempre el prefijo **static**.

```
protocol UnProtocolo {  
    var unaPropiedad : UnTipo {get}  
    static 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.
 - (Nota: Desde Swift 2 se pueden proporcionar implementaciones por defecto)
 - Los métodos de clase usan siempre el prefijo **static**.
 - Si el método es mutante, usar el prefijo **mutating**.

```
protocol UnProtocolo {  
    func unMetodo(a: Int) -> Bool  
    static 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 sobrescribir).
- Si la implementación del inicializador requerido por el protocolo, además sobrescribe 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 los nombres de los protocolos separados por el operador **&**.

```
var x: UnProtocolo & OtroProtocolo
```

```
func unafuncion(v: UnProtocolo & OtroProtocolo) { ... }
```

```
func otraFuncion() -> UnProtocolo & OtroProtocolo { ... }
```

- Una composición es otro tipo, y puede usarse igual que cualquier otro tipo.
- Nuevo en Swift 4: Puede especificarse un tipo como la composición de una clase con cualquier número de protocolos.

```
var x: UnaClase & UnProtocolo & OtroProtocolo
```

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.

```
protocol Domesticable { . . . }  
class Animal {}  
class Dog : Animal, Domesticable { . . . }  
class Lion : Animal {}  
  
var a : Animal = ???  
  
if a is Domesticable { . . . }  
  
if let d = a as? Domesticable { . . . }
```

Protocolos: Requisitos Opcionales

- Los requisitos opcionales no tienen que implementarse obligatoriamente por los tipos que adopten el protocolo.
 - Nota:
 - Se necesitan para crear código que interopere con código en Objective-C.
 - Los protocolos y los requisitos opcionales deben marcarse en el atributo **@objc**.
 - Solo pueden ser adoptados por clases que hereden de clases Objective-C o de otras clases @objc.
- 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 {
    @objc optional var growl: String {get}
    @objc optional func talk() -> String
}
class Dog : Domesticable {}

var dom: Domesticable = ???

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

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 sobrescribir la implementación por defecto proporcionada por el protocolo.

- Creamos una extensión del protocolo **Collection** para añadir el método **count(if:)** con una implementación por defecto:

```
extension Collection {  
  
    func count(if condition: Self.Iterator.Element -> Bool)  
                -> Int {  
  
        var n = 0  
        for value in self where condition(value) {  
            n += 1  
        }  
        return n  
    }  
}
```

```
var c = [1,2,3,4,5,6,7].count(if: {$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 Collection
  where Self.Iterator.Element: CustomStringConvertible {

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

Genéricos

Genéricos

- Los genéricos permiten crear funciones y tipos que pueden trabajar / usar / contener / etc. con cualquier tipo.
- En Swift 4 se añaden subscript genéricos.
- Ejemplos:
 - El tipo **Array** permite puede crearse para almacenar datos cualquier 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>(a: inout T, b: inout 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 en los tipos en los parámetros y en el cuerpo de la función.

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

Extender Tipos Genéricos

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

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

```
s.top() // 2
```

Subscripts Genéricos (Solo Swift 4)

- Pueden usarse genéricos con el tipo devuelto y con el tipo índice de los subscripts.

```
// almacen de cualquier tipo de dato
struct Store {
    private var data: [Any] // array de cualquier cosa
    init(data: [Any]) {
        self.data = data
    }

    // Subscripts para acceder a los datos por su indice.
    EN LAS SIGUIENTES TRANSPARENCIAS

    // Subscripts para obtener los datos en varios indices.
    EN LAS SIGUIENTES TRANSPARENCIAS
}
var t = Store(data: [1, true, "hola"])
var a: Int = t[0] // Primer elemento.
var b: Bool = t[1] // Segundo elemento.
var c = t[[0,2]] // Los indices son un array - [1, "hola"]
var d = t[1...2] // Los indices son un rango ... - [true, "hola"]
var e = t[1..<3] // Los indices son un rango ..< - [true, "hola"]
```

- Subscripts para acceder a un dato por su índice:

- Sin genéricos. Creamos un subscript para cada tipo para no tener que hacer castings:

```
subscript(n: Int) -> Int {  
    return data[n] as! Int  
}  
subscript(n: Int) -> Bool {  
    return data[n] as! Bool  
}  
subscript(n: Int) -> String {  
    return data[n] as! String  
}
```

- Usando genéricos:

```
subscript<T>(n: Int) -> T {  
    return data[n] as! T  
}
```

- Subscripts para acceder a los datos en varios índices:

- Sin genéricos. Indicar los índices como un array o rangos:

```
subscript(keys: [Int]) -> [Any] {  
    return keys.map {data[$0]}  
}
```

```
subscript(keys: CountableClosedRange<Int>) -> [Any] {  
    return Array(data[keys])  
}
```

```
subscript(keys: CountableRange<Int>) -> [Any] {  
    return Array(data[keys])  
}
```

- Usando genéricos:

```
subscript<Keys: Sequence>(keys: Keys) -> [Any]  
    where Keys.Iterator.Element == Int {  
    return keys.map {data[$0]}  
}
```

Restricciones

- Pueden imponerse restricciones sobre los tipos que pueden usarse con una función genérica, con un tipo genérico, o con un subscript.
- 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 **associatedtype** para realizar la misma tarea.
- ¿Cómo funciona?:
 - Al definir un protocolo se usa **associatedtype** 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 {
    associatedtype 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, at: 0) }
}

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

```

Cláusulas **where**

- Pueden definirse requisitos sobre los tipos asociados de un protocolo usando cláusulas **where**.

- La cláusula **where** se pone al final de la signatura, antes del body.
- 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>(q1: Q1, q2: Q2)
    where Q1.T == Q2.T {
    if let x = q1.get() {
        q2.put(x)
    }
}
var d1 = Dispatcher(), d2 = Dispatcher()
d1.put(Task(name="Prueba"))
transfer(q1: d1, q2: 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 elementos del mismo tipo.

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 abierto:** Se usa el modificador **open**. El acceso es igual que público, pero además una clase se puede derivar en otro módulo, y un método se puede sobrescribir en otro módulo.
 - **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 en el fichero:** Se puede acceder a los elementos con el modificador **fileprivate** solo desde el mismo fichero en el que están definidos. No se pueden usar desde otros ficheros ni desde otros módulos.
 - **Acceso privado:** Se puede acceder a los elementos con el modificador **private** solo desde el mismo ámbito léxico en el que están definidos (entre los { y } donde está definido). Con Swift 4, también se puede acceder a estos elementos desde extensiones del tipo definidas dentro del mismo fichero swift. No se pueden usar desde otros ficheros, módulos o ámbitos.
 - Si la definición **fileprivate** se hace en el ámbito global (el más externo del fichero), se puede acceder al elemento desde cualquier punto del fichero.

```
public class Dog {  
    internal let speed = 0  
  
    var name = "Coco" // por defecto es internal  
  
    private var age = 1  
  
    fileprivate func run() { }  
  
}
```

- El nivel de acceso de los diferentes elementos de un programa se puede especificar con los modificadores **public**, **open**, **internal**, **fileprivate** 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)**, **fileprivate(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, subscriptos*).
 - 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 sobrescribirse 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 explícitamente 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 más restrictivo que el del tipo en el que se basa.
- etc...

Operadores Avanzados

Operadores de Bits

- \sim es un NOT.
- $\&$ es el AND
- $|$ es el OR
- \wedge es el XOR
- \ll es el desplazamiento a la izquierda
- \gg es el desplazamiento a la derecha

Operadores con Desbordamiento

- Son: $\&+$ $\&-$ $\&*$
 - Si se produce desbordamiento el programa no se muere.

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

```
prefix func ++ (v: inout 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 crear sus propios 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 se especifica a que grupo de precedencia pertenece.

```
prefix operator +++ {}
```

```
infix operator +- : AdditionPrecedence
```

- Un grupo de precedencia especifica el tipo de asociatividad para los operadores infijos, si el nuevo operador se usa para realizar asignaciones, y la relación con otras precedencias (mayor que otra, menor que otra, ...).
- También pueden crearse nuevos grupos de precedencias.

- Ejemplo de implementación de nuevos operadores:

```
prefix operator +++
```

```
infix operator +- : AdditionPrecedence
```

```
prefix func +++ (v: inout 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 ahora Vector(x:4, y:6)  
a +- b    // Vector(x:10, y:4)
```

Los grupos de precedencias predefinidos son:

```
BitwiseShiftPrecedence > MultiplicationPrecedence (left) >
AdditionPrecedence (left) > RangeFormationPrecedence > CastingPrecedence
> NilCoalescingPrecedence > ComparisonPrecedence >
LogicalConjunctionPrecedence (left) > LogicalDisjunctionPrecedence (left)
> DefaultPrecedence > TernaryPrecedence (right) > AssignmentPrecedence
(right, assignment) > FunctionArrowPrecedence (right) > [nothing]
```

Si no se especifica un grupo de precedencia al definir un operador se usa **DefaultPrecedence**.

Cada grupo de precedencia puede opcionalmente definir su relación con otros grupos usando **higherThan**. Para relacionar con operadores definidos en otros módulos puede usarse también **lowerThan**.

Cada grupo de precedencia puede opcionalmente definir su tipo de asociatividad, que puede ser **left** o **right**.

Para los operadores que pueden realizar asignaciones en cadenas de opcionales se usa

assignment:

```
precedencegroup AssignmentPrecedence {
    associativity: right
    assignment: true
    higherThan: FunctionArrowPrecedence
}
```

Para crear un grupo de precedencia personalizado:

```
precedencegroup HighPrecedence {
    higherThan: BitwiseShiftPrecedence
}
precedencegroup LeftAssociativePrecedence {
    associativity: left
}
```

Otros Temas

Atributos

- Proporcionar más información sobre una declaración o un tipo.
 - available, discardableResult, objc, nonobjc, UIApplicationMain, NSCopying, IBAction, IBOutlet, IBDesignable, IBInspectable, autoclosure, ...
- Ver la documentación para más detalles.
 - Libro: The Swift Programming Language.

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**, **watchOS** y **tvOS**) y versiones mínimas, terminando con ***** para indicar otras plataformas).

Expresión # selector

- Esta expresión devuelve el selector a usar para referirse a un método, o los método de acceso (getter y setter) de una propiedad disponible de Objective-C.
 - Esta expresión devuelve una instancia del tipo SELECTOR:
- En vez de usar un String usamos esta expresión.

```
#selector(unaClase.unMetodo)
```

```
#selector(unaClase.unMetodo(arg1:arg2:))
```

```
#selector(getter: instancia.propiedad)
```

```
#selector(setter: instancia.propiedad)
```

- Puede usarse el operador **as** para evitar ambigüedades entre métodos con la misma signatura.
- Permite comprobar que exista la propiedad, que sea accesible, refactorizar, no es necesario saber cuál es el nombre real en Objective-C, ...

KeyPath

- Es un tipo de datos cuyos valores describen una ruta desde un tipo base hasta el valor de alguna propiedad.
- Antes de Swift 3, un KeyPath era un literal de tipo String.
 - No se podían detectar fallos en tiempo de compilación.
- En Swift 3 se introdujo la expresión `#keyPath`.
 - Estas expresiones devuelven el string que representa una ruta.

`#keyPath(ClaseBase.unaPropiedad)`

- En Swift 4 se introducen literales KeyPath.
 - Formados por un backslash, seguido del tipo base y de las propiedades de la ruta, separando con puntos.

`\ClaseBase.unaPropiedad`

- Puede omitirse la clase base cuando pueda inferirse su valor:

```
\Person.name
```

```
\.name
```

- Pueden componerse añadiendo más propiedades:

```
\Person.car.model
```

```
let kp1 = \Person.car
```

```
let kp2 = kp1.appending(path: \.model)
```

- Puede usarse Optional Chaining:

```
\Person.girlfriend?.name
```

- Indirección con subscripts:

```
\Person.friend[0]
```

- Uso directo de subscripts:

```
\Data.[0]
```

- Se accede a los datos usando un subscript que usa la etiqueta keyPath:

```
let a = peter[keyPath: \Person.age]
```

```
peter[keyPath: \Person.age] = 33
```

- El tipo `KeyPath` es un genérico:

`KeyPath`*<TipoBase, TipoDeUltimaPropiedadDeLaRuta>*

- Jerarquía de Tipos:

`AnyKeyPath`

`PartialKeyPath`*<TipoBase>*

`KeyPath`*<TipoBase, TipoDePropiedad>*

`WritableKeyPath`*<TipoBase, TipoDePropiedad>*

`ReferenceWritableKeyPath`*<TipoBase, TipoDePropiedad>*

- Los dos últimos para cambiar tipo base de tipo valor o referencia.

Compilación Condicional

- Usando la directivas de compilación:

```
#if Condicion_Compilacion
    sentencias
#elseif Condicion_Compilacion
    sentencias
#else
    sentencias
#endif
```

- Condiciones de compilación:

- Pueden incluir:

- **true**, **false**.
- identificadores usados en la opción **-D** del compilador.
 - **debug** está definida solo en modo depuración.
- condiciones sobre la plataforma

```
os(OSX|iOS|watchOS|tvOS|Linux)
arch(i386|x86_64|arm|arm64)
swift(>=version)
```
- **()**, **!**, **&&**, **||**.

Expresiones

- Expresiones útiles para depurar:
 - **#file** - Nombre del fichero que la contiene.
 - **#column** - Número de columna en el código.
 - **#function** - Nombre de la función que la contiene.
 - **#line** - Número de línea en el código.

