

ebook

INTRODUCCIÓN A SWIFT

EFFECTO/APPLE



WWW.EFFECTOAPPLE.COM

Por Luis Rollón.
Creador de www.efectoapple.com

WWW.EFECTOAPPLE.COM

ÍNDICE

WWW.EFECTOAPPLE.COM

1. Bienvenido
2. Objetivo del Libro
3. Introducción
4. Todo es un Objeto
5. Variables y Constantes
6. Inferencia de Tipos
7. Typealias
8. Control de Flujo
9. Aclaración sobre las Colecciones
10. Arrays
11. Diccionarios
12. Sets
13. Tuplas
14. Pattern Matching
15. Funciones
16. Closures
17. Opcionales
18. Clases
19. Structs
20. Enums
21. Extensiones
22. Protocolos
23. Genéricos
24. Despedida

BIENVENIDO

WWW.EFECTOAPPLE.COM

Lo primero que me gustaría hacer es darte la bienvenida.

Tanto si eres Desarrollador iOS como si estás pensando en serlo, estoy seguro que este ebook te será muy útil.

Espero que puedas encontrar mucho valor en las siguientes páginas. He puesto todo de mi parte para que así sea.

El libro que tienes delante está compuesto de decenas de ejemplos realizadas en el Playground de Xcode.

Si no estás familiarizado con el concepto de Playground, simplemente debes saber que es un espacio dentro de Xcode perfecto para realizar pruebas y comprender diferentes conceptos de programación.

Una de sus ventajas es la posibilidad de ver en tiempo real los resultados del código que vamos desarrollando. En la parte derecha del editor se irán mostrando las salidas que nuestras líneas de código van generando.

Para acceder al Playground desde Xcode simplemente deberás pulsar en el menú **File > New...** y hacer click en **Playground...**

Xcode te pedirá un nombre para tu archivo y te preguntará donde quieres guardarlo.

Una vez que hayas seleccionado la ubicación donde guardarla, podrás comenzar a escribir código en tu Playground.

Este libro está lleno de decenas de ejemplos realizadas en Playground. Te recomiendo que mientras vas avanzando en los conceptos que vamos presentando, utilices tu propio Playground para realizar los ejercicios que van apareciendo.

Esto te permitirá comprender al 100% las características de Swift y aprovechar al máximo este ebook.

Recuerda, el conocimiento no consiste en saber sino en **SABER HACER**.

El objetivo de este ebook es el de sentar las bases fundamentales de la tercera versión de Swift desarrollada por Apple.

Puede que tengas experiencia en desarrollos con Swift o puede que nunca hayas trabajado con este lenguaje.

En cualquiera de los dos casos, estoy casi seguro al 100% que este ebook puede enseñarte muchas cosas.

Lo que verás a través de las próximas páginas serán explicaciones claras y prácticas de los conceptos fundamentales sobre los que se asienta Swift.

Comenzaremos por conceptos tan básicos como Constantes y Variables y continuaremos hasta llegar a aspectos avanzados como: Genéricos, Extensiones y Protocolos.

Si comienzas de cero con este lenguaje te recomiendo que comiences desde aquí y vayas trabajando desde los conceptos iniciales. A pesar de que se comience desde un nivel básico, Swift posee algunas particularidades que te recomiendo que conozcas.

Si ya tienes experiencia como desarrollador, puede que prefieras acceder rápidamente a algún concepto en concreto que no tengas claro del todo. En este caso, puedes utilizar el Índice para aprender algún aspecto concreto de Swift.

En cada uno de los conceptos sobre los que vamos a trabajar, realizaremos ejemplos prácticos que te servirán para asentar conocimientos. No tiene sentido explicar teoría sin poder ver su aplicación práctica.

No perdamos más tiempo y...

¡COMENCEMOS!

OBJETIVO DEL LIBRO

WWW.EFECTOAPPLE.COM

Es importante dejar bien claro que es y que no es este libro.

Empecemos por repasar lo que NO es:

- No se trata de un libro donde veremos de inicio a fin todo el lenguaje Swift
- No se trata de un libro de referencia donde mostraremos todas las clases y todos los métodos del lenguaje
- No se trata de un libro para alguien que no tenga unos conocimientos mínimos de programación orientada a objetos
- No se trata de un libro que te va a enseñar a desarrollar aplicaciones iOS

El objetivo de este libro es servir como introducción al lenguaje Swift.

A lo largo de sus páginas repasaremos de forma básica y siempre práctica, utilizando decenas de ejemplos, los aspectos más importantes de Swift.

Si tienes unos conocimientos mínimos de programación y estás interesado en saber en qué consiste Swift este es tu libro y estoy seguro que lo vas a disfrutar.

INTRODUCCIÓN

WWW.EFECTOAPPLE.COM

Antes de comenzar, es importante tener unas pequeñas nociones de que es Swift:

- Swift fue anunciado en 2014.
- Puede utilizarse para desarrollar aplicaciones para iOS, macOS, tvOS y watchOS.
- Actualmente Apple ha lanzado la versión 3.1 de este lenguaje.
- Es un lenguaje con una sintaxis moderna, sobre todo si la comparamos con Objective-C.
- Puede integrarse con C y con Objective-C.

Swift se trata de un lenguaje muy diferente a Objective-C. Se podría decir que son lenguajes opuestos. Estas son algunas de sus características:

- Tiene tipado estricto
- No existen los ficheros de cabecera, por lo que se acabaron los ficheros .h
- Cuando accedes a nil se cae la app. En Objective-C ya sabes que esto no pasaba.
- Desaparecen los puntos y coma al final de cada linea

Una vez que ya sabemos algo de nuestro nuevo mejor amigo, en las siguientes secciones podremos ir conociéndolo más a fondo.

TODO ES UN OBJETO

WWW.EFECTOAPPLE.COM

Una característica especial de Swift es que todo es un objeto. En la mayoría de los lenguajes se diferencia entre tipos primitivos y objetos. En Swift no pasa eso.

En Swift no existen los tipos primitivos.

Si es cierto, que existen excepciones. Puede que algunos elementos que utilizemos en Swift no sean exactamente objetos sino que pueden ser **Tipos Agregados**. Este es un concepto que revisaremos más adelante, por ahora no te preocupes por él.

Por ejemplo, para representar números en Swift podemos utilizar los tipos Int, Double o Float. Estos tipos de datos, son clases, con sus métodos, propiedades de instancia y de clase. Son exactamente iguales que cualquier otro objeto. Por tanto, no se tratan de tipos primitivos, como podríamos utilizar en Objective-C.

En estos dos ejemplos, puedes ver como las clases Int y Double tienen sus propios métodos de clase:

```
//Función de Int que nos permite obtener el número máximo que podemos almacenar  
//con la clase Int  
Int.max  
  
//Función de Double para obtener el valor absoluto de un número  
Double.abs(-3)
```

VARIABLES y CONSTANTES

Para declarar una variable en Swift, usamos la palabra reservada **var**.

La sintaxis sería la siguiente:

```
var var_name : var_type = var_value
```

Para declarar una constante usamos la palabra reservada **let**.

La sintaxis sería la siguiente:

```
let const_name : const_type = const_value
```

Aquí tienes dos ejemplos de declaraciones de variable y constante respectivamente:

```
//Declaración de una variable  
var age : Double = 32  
  
//Declaración de una constante  
let name : String = "Tyrion"
```

En estos ejemplos, podemos apreciar la diferencia respecto a Objective-C: Especificamos el tipo de la variable después del nombre, no delante.

Tal vez te hayas dado cuenta que la declaración de variables y constantes es redundante. En Objective-C también se producía esta redundancia.

¿A qué nos referimos con redundante?

Lo veremos en el siguiente concepto.

INFERENCIA DE TIPOS

WWW.EFECTOAPPLE.COM

Como hemos comentado en el anterior ejemplo, en la declaración de nuestras variables y constantes se produce una redundancia:

```
//Declaración de una variable  
var age : Double = 32  
  
//Declaración de una constante  
let name : String = "Tyrion"
```

Voy a explicar a lo que me refiero.

Si a nuestra variable `age` le asignamos el valor `32`, el compilador ya debería ser capaz de darse cuenta de que estamos declarando una variable que almacenará un número. Puede que no sepa si es un entero, un `double`, pero si que sabe que es un número, por el valor `32` que le estamos dando.

Lo mismo y de forma más clara, pasa con la constante `name`. Al asignarle el valor `"Tyrion"`, el compilador es capaz de deducir que el tipo de esta constante es una cadena (`String`).

Por tanto, si nosotros indicamos el tipo de nuestras variables o constantes, estamos realizando un trabajo repetitivo y que no es necesario que hagamos. El compilador puede trabajar sin problema sin tener esa información extra.

Y eso es precisamente lo que hace Swift. No le hace falta que aportemos la información de tipo, el compilador es capaz de deducir esa información. A esta deducción se le denomina **Inferencia de tipo**.

Veámoslo con un nuevo ejemplo:

```
//Declaración de una variable sin especificar su tipo  
var age = 32  
  
//Declaración de una constante sin especificar su tipo  
let name = "Tyrion"
```

Ambas opciones al declarar una variable son válidas. Lo recomendable es no especificar el tipo, ya que si el compilador es capaz de deducirlo, ¿para qué vamos a perder el tiempo escribiéndolo?

Por tanto, la **Inferencia de Tipo** es la característica del compilador que le permite deducir el tipo de una variable o constante a partir del valor que le estamos asignando al inicializarla.

TYPEALIAS

WWW.EFECTOAPPLE.COM

El **Typealias** nos permite asignar un "sobrenombre" a un tipo de dato que ya existe.

Es como darle un nuevo nombre a un tipo existente.

Es similar al `typedef` en C.

Puede que esto te parezca algo que no tiene mayor importancia, sin embargo te será muy útil más adelante.

La sintaxis es la siguiente:

```
typedef new_name = old_name
```

Aquí tienes un ejemplo muy claro:

```
//A la clase Int le asignamos el typealias Integer
typealias Integer = Int

//Una vez creado ya podemos utilizar el typealias como si fuera la propia clase
var length : Integer = 26
```

CONTROL DE FLUJO

WWW.EFECTOAPPLE.COM

En esta sección veremos como controlar el flujo de nuestros programas en Swift.

Cuando hablamos de controlar el flujo, nos referimos a poder modificar el orden de ejecución de las instrucciones que forman nuestro código.

Existen dos formas de controlar el flujo en un programa:

- Mediante Condicionales
- A través de Bucles

En los condicionales revisaremos los siguientes:

- If
- If-else
- Switch

Por otro lado, en la parte de bucles, veremos:

- For-in
- While
- Repeat-While

Comencemos entonces por los condicionales.

Veamos como utilizar la sentencia If en swift, con un ejemplo muy sencillo:

```
var number = 12

if number > 10 {
    print("number almacena un valor mayor de 10")}
```

Hemos creado una variable *number* y le hemos asignado el valor 12. Después hemos utilizado la sentencia If para que, en el caso de que el valor de *number* sea mayor que 10, muestre por pantalla ese mensaje de texto.

Si escribes este código en tu playground verás como, al cumplirse la condición, se muestra por pantalla dicho mensaje:

```
number almacena un valor mayor de 10
```

Si modificáramos el valor de la variable *number* y le asignáramos por ejemplo, un 8, no se mostraría nada por pantalla, puesto que no cumpliría la condición que evaluamos en la sentencia If.

Si quisiéramos que otro mensaje apareciera en pantalla cuando el valor de *number* sea inferior a 10, ampliaríamos nuestra sentencia If, convirtiéndola en un If-else:

```
145 var number = 8
146
147 if number > 10 {
148     print("number almacena un valor mayor de 10")
149 }else {
150     print("number almacena un valor menor de 10")
151 }
```

number almacena un valor menor de 10

Como puedes ver, hemos cambiado el valor de *number* por un 8 y ahora se muestra en pantalla el mensaje de texto que corresponde a la sentencia *else*.

Con estas dos sentencias podemos evaluar condiciones y modificar el flujo de nuestros programas en base a nuestras necesidades.

Existe otra sentencia que nos permite evaluar una gran cantidad de condiciones sin necesidad de utilizar diferentes Ifs anidados. Evidentemente nos estamos refiriendo a un *Switch*.

Veamos un ejemplo muy sencillo, que nos permitirá entender la sintaxis de un *switch*:

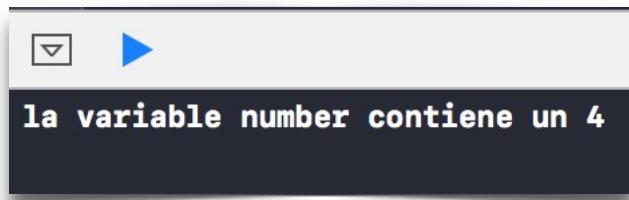
```
var number = 4

switch number {
case 0:
    print("la variable number contiene un 0")
case 1:
    print("la variable number contiene un 1")
case 2:
    print("la variable number contiene un 2")
case 3:
    print("la variable number contiene un 3")
case 4:
    print("la variable number contiene un 4")
default:
    print("la variable number contiene un número mayor de 5")
}
```

Hemos creado una variable *number*, en la que hemos almacenado el valor 4.

Hemos evaluado los casos en los que *number* pueda almacenar valores del 0 al 4. Además, hemos utilizado la opción *default* para especificar que la variable *number* almacena un número mayor de 5, si se da el caso.

Como ahora mismo *number* vale 4, se mostrará por pantalla el siguiente mensaje de texto:



Si modificáramos el valor de la variable, evidentemente el mensaje por pantalla cambiaría.

Es importante mencionar, que el switch en swift, debe contemplar siempre todos posibles casos. Para esto, como has podido ver, es muy útil la sentencia *default*.

Una vez que hemos revisados la parte de los condicionales, pasemos a los bucles en swift.

Como hemos comentado antes, vamos a revisar 3 bucles diferentes.

Comencemos por el primero:

Bucle For-In:

El bucle For-In permite repetir una porción de código un determinado número de veces.

Este trozo de código se repetirá en función de una progresión, un rango o una colección de datos.

Este bucle es probablemente el más utilizado en Swift.

Su principal función es la de recorrer colecciones de datos, permitiéndonos trabajar con cada uno de los elementos de forma individual.

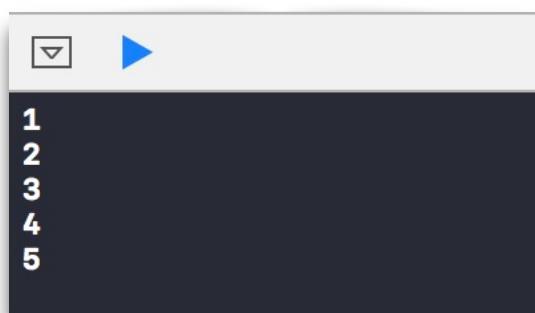
Veamos un ejemplo muy sencillo, para entender su sintaxis:

```
for indice in 1...5{  
    print(indice)  
}
```

Hemos creado un rango del 1 al 5 (1...5) que nos permitirá hacer que el bucle se ejecute 5 veces.

Además, en cada una de esas repeticiones, la variable índice tomará un valor diferente del rango, desde el 1 hasta el 5.

Por lo que, esta porción de código mostrará por pantalla lo siguiente:

A screenshot of an Xcode interface showing a terminal window. The window has a dark background and a light gray header bar with a downward arrow icon and a blue play button icon. In the terminal area, the numbers 1, 2, 3, 4, and 5 are displayed vertically, each on a new line, representing the output of the code above.

```
1  
2  
3  
4  
5
```

Como te digo, si trabajas con swift, utilizarás constantemente el bucle For-In para recorrer colecciones.

Pasemos al siguiente bucle de nuestra lista: While.

El bucle while nos permite repetir la ejecución de un trozo de código mientras se cumpla una determinada condición.

Pasemos a ver su funcionamiento a través de un ejemplo similar al que vimos en el bucle anterior:

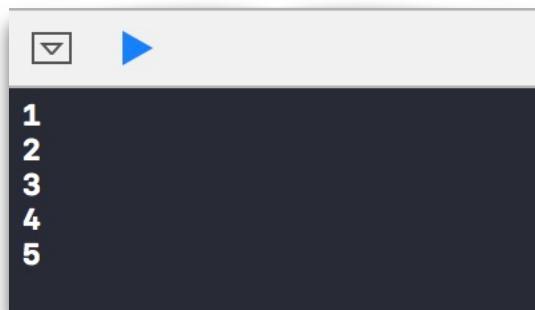
```
var indice = 1
while indice <= 5 {
    print(indice)
    indice = indice + 1
}
```

Hemos creado una variable indice y le hemos asignado el valor 1.

Después hemos utilizado el bucle while para que ejecute el código que tiene en su interior mientras la variable indice sea menor o igual que 5.

En el interior del bucle, mostramos por pantalla el valor de indice y además incrementamos en 1 su valor. Es decir, si en su primera iteración, indice vale 1, en la segunda valdrá 2 y así sucesivamente.

A través de este bucle while, conseguimos exactamente el mismo resultado que obtuvimos en el ejemplo anterior con el bucle For-In:



El bucle *while* se repetirá las veces necesarias hasta que la condición deje de cumplirse.

Por último para finalizar esta sección, hablaremos del bucle *repeat-while*.

Su funcionamiento será exactamente igual que el bucle *while* pero con una diferencia.

Mientras el bucle *while* puede darse el caso que no se ejecute ninguna vez (Cuando en su primera iteración no se cumpla la condición), el bucle *repeat-while* se ejecutará como mínimo una vez.

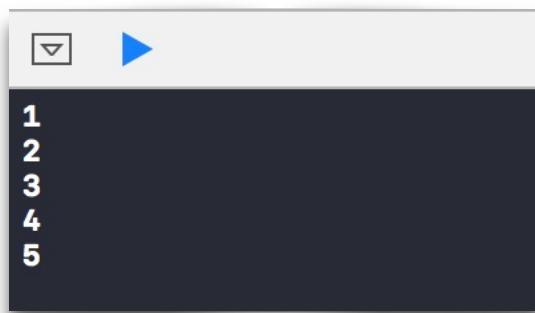
Expliquemos esto con otro ejemplo:

```
var indice = 1
repeat{
    print(indice)
    indice = indice + 1
}while indice <= 5
```

Hemos elegido el mismo tipo de ejemplo que en los 2 bucles anteriores.

El código incluido dentro de la cláusula *repeat* *ll* se repetirá mientras se cumpla la condición que aparece detrás del *while*.

Evidentemente obtendremos la misma salida por consola:



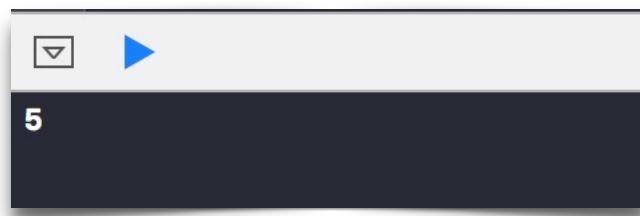
```
1
2
3
4
5
```

Sin embargo, si te das cuenta, la evaluación de la condición está situada después del bloque de código, por lo que aunque en la primera iteración del bucle, esta condición no se cumpla, la porción de código del bucle ya se habrá ejecutado como mínimo una vez.

Podemos comprobar esto, sustituyendo el valor de la variable indice por un 5:

```
var indice = 5
repeat{
    print(indice)
    indice = indice + 1
}while indice <= 5
```

Y observando el resultado por pantalla de este código:



Puedes observar, que a pesar de que *indice* vale 5 y por tanto no cumple la condición del bucle, como mínimo se ejecuta una vez.

Por tanto, la única diferencia entre el bucle *while* y el *repeat-while* es que el primero puede que no llegue a ejecutar el código incluido en él ni una sola vez, mientras que en el segundo, como mínimo se ejecutará una vez.

ACLARACIÓN SOBRE LAS COLECCIONES

WWW.EFECTOAPPLE.COM

Seguro que ya sabes que cuando nos referimos a Colecciones, hablamos de estos 3 elementos:

- Arrays
- Diccionarios
- Sets

En Objective-C se puede utilizar una colección para almacenar diferentes tipos de datos dentro de ella. Es decir, dentro de un array por ejemplo, puedes almacenar un NSString, un NSNumber y un NSDictionary.

Es decir, en Objective-C, los elementos almacenados dentro de una colección no tienen porque ser del mismo tipo.

Esto no es así en Swift. En Swift, dentro de una colección únicamente podremos almacenar objetos del mismo tipo. Por ejemplo, si guardamos un String en un Array, el resto de objetos a almacenar deben ser también Strings. Si almacenamos un Integer y un String dentro de un Diccionario, el resto de elementos deben ser Integer y String.

A pesar de esto, existe una posibilidad de poder almacenar diferentes tipos de objetos dentro de una misma colección en Swift. Para ello habrá que importar el framework Foundation en nuestras clases. Importando este framework, podremos utilizar las típicas colecciones de Cocoa:

- NSArray
- NSDictionary
- NSSet

que si que nos permiten almacenar objetos heterogéneos en la misma colección.

Si lo que realmente queremos es utilizar Swift puro, no importaremos Foundation y utilizaremos las colecciones de Swift:

- Array
- Dictionary
- Set

y habrá que tener en cuenta que solo podremos utilizarlas para almacenar objetos del mismo tipo.

En todos los ejemplos de este eBook utilizaremos swift puro, por lo que no importaremos Foundation en nuestras clases.

Repasemos entonces como trabajar con Arrays, Diccionarios y Sets en Swift.

ARRAYS

WWW.EFECTOAPPLE.COM

Un array es una colección ORDENADA de datos del mismo tipo.

Para representar un array, escribiremos un conjunto de elementos separados por comas y acotados por corchetes.

Es importante resaltar, que un array es un tipo de dato por valor y no por referencia. Es decir, si asignamos un array a otro, siempre se realizará una copia del primer array.

Sin embargo, si lo que almacenamos en el array es un valor por referencia, aunque dicho valor se copie en otro array, seguirá siendo un dato por referencia y tanto el valor almacenado en el array inicial como el almacenado en la copia serán el mismo objeto.

Una vez aclarado esto, veamos las diferentes opciones que tenemos para declarar e inicializar un array:

```
//Declaración de un Array de tipo String, especificando su tipo
let casaStark: [String] = ["Eddard", "Rob", "Catelyn", "Sansa"]

//Declaración de un Array sin especificar su tipo
let familiaSimpson = ["Homber", "Marge", "Bart", "Lisa", "Maggie"]

//Declaración de un Array de tipo Int
var numerosLost = [4, 8, 15, 16, 23, 42]
```

Como puedes ver en los ejemplos, podemos declarar e inicializar un array, especificando su tipo.

También podemos dejar que sea el propio compilador quien infiera el tipo del array. En el segundo ejemplo, queda claro que no determinamos de ninguna forma que nuestro array es de tipo string y es el propio compilador quien obtiene esa información a partir de los datos que hemos introducido en el array.

En el último ejemplo, ves como hemos declarado un array de tipo Int de forma sencilla.

Pero, que pasaría si en lugar de declarar e inicializar un array, ¿simplemente queremos declararlo y dejarlo vacío para poder inicializarlo más adelante?

Haríamos lo siguiente:

```
//Declaración de un array vacío
var arrayVacio = [String]()

//Inicialización del array vacío
arrayVacio = ["Elemento 1", "Elemento 2", "Elemento 3", "Elemento 4", "Elemento 5"]
```

Como ves, a la hora de declarar un array vacío, si que es imprescindible especificar su tipo.

Una vez que lo hemos declarado podremos inicializarlo con elementos del tipo que hemos especificado en su declaración.

En este último ejemplo, hemos creado un array vacío de tipo String y lo hemos llenado con elementos de tipo String.

Como probablemente sepas, cuando creamos un array y lo inicializamos, se le asigna un índice a cada uno de los elementos almacenados en el array, comenzando desde cero.

De esta forma, si queremos acceder a cualquier de los elementos almacenados en dicho array, simplemente tendremos que especificar el nombre del array y el índice entre corchetes:

<pre>//Declaración de un array vacío var arrayVacio = [String]() //Inicialización del array vacío arrayVacio = ["Elemento 1", "Elemento 2", "Elemento 3"] arrayVacio[2]</pre>	<pre>[]</pre>
	<pre>["Elemento 1", "Elemento 2", "Elemento 3"]</pre>
	<pre>"Elemento 3"</pre>

En este caso, hemos accedido al último elemento del array a través del índice 2.

Existen otras operaciones comunes cuando trabajamos con arrays. Por ejemplo, añadir elementos al array, borrar elementos o mostrar el contenido de un array.

En el siguiente ejemplo, tienes estas 3 operaciones:

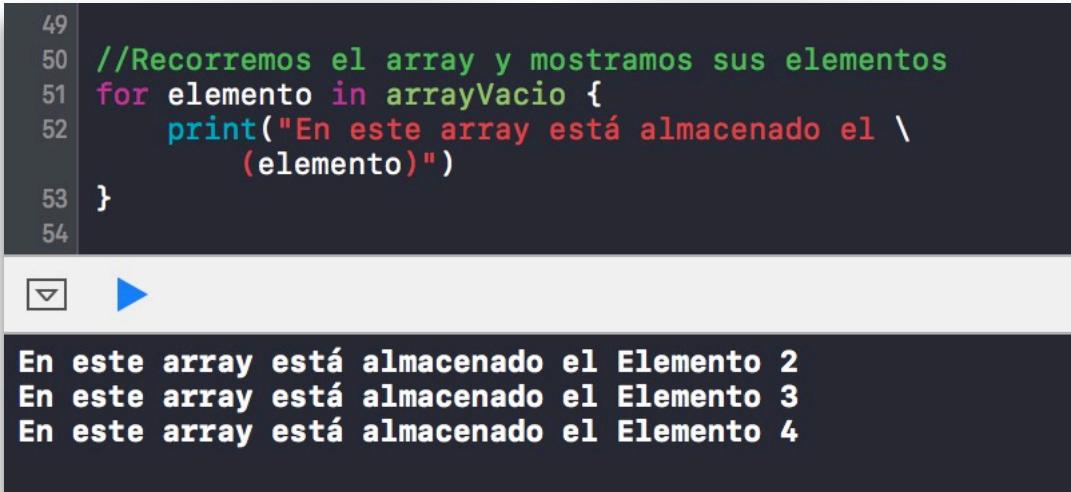
//Añadiendo un elemento al array arrayVacio.append("Elemento 4") //Eliminando un elemento del array arrayVacio.remove(at: 0) //Mostrando el contenido del array print(arrayVacio)	["Elemento 1", "Elemento 2", "Elemento 3", "Elemento 4"] "Elemento 1" "["Elemento 2", "Elemento 3", "Elemento 4"]\n"
--	--

Podemos añadir elementos a través del método `append()`, eliminar elementos usando `remove()`, seguido del índice del elemento que queramos eliminar y por supuesto, para mostrar el contenido de un array usaremos el método `print()`.

Probablemente la operación más importante que harás con un array sea recorrerlo. Es decir, ser capaz de pasar por todos y cada uno de los elementos incluidos dentro de un array y poder trabajar de forma individual con ellos.

La forma mas común de hacer esto es utilizando el bucle `for-in`.

Veamos en el siguiente ejemplo, como recorrer un array y mostrar por pantalla cada uno de los elementos contenidos en él.



The screenshot shows a Swift code editor with the following code:

```
49  
50 //Recorremos el array y mostramos sus elementos  
51 for elemento in arrayVacio {  
52     print("En este array está almacenado el \  
      (elemento)")  
53 }  
54
```

Below the code, there are two navigation icons: a downward arrow and a right-pointing triangle. The output window displays the following text:

En este array está almacenado el Elemento 2
En este array está almacenado el Elemento 3
En este array está almacenado el Elemento 4

Esta es la forma más común de recorrer un array y que utilizarás muchas veces si terminas trabajando con Swift.

DICCIONARIOS

WWW.EFECTOAPPLE.COM

Un diccionario es una colección de datos NO ORDENADA, que se almacena en parejas clave-valor.

Cuando trabajamos con arrays, es el propio sistema el que añade los índices de ordenación a los elementos que vamos añadiendo, sin embargo en los diccionarios, somos nosotros quienes establecemos tanto la clave ó índice de búsqueda como el propio elemento.

Por eso decimos que se almacenan parejas clave-valor.

Tanto la clave como el valor pueden ser cualquier tipo de dato.

Para declarar e inicializar un diccionario, cada pareja clave-valor irá separada del resto por comas (,) y la clave irá separada del valor por dos puntos (:). Además todo el conjunto de elementos irán entre corchetes ([]).

Veamos mejor con ejemplos, las diferentes opciones que tenemos para declarar e inicializar un diccionario:

```
//Declaración de un Diccionario de tipo Int:String, especificando su tipo
let casaStark: [Int:String] = [1:"Eddard", 2:"Rob", 3:"Catelyn", 4:"Sansa"]

//Declaración de un Diccionario de tipo Int: String sin especificar su tipo
let familiaSimpson = [1: "Homer", 2: "Marge", 3: "Bart", 4: "Lisa", 5: "Maggie"]

//Declaración de un Diccionario de tipo String: String
var Starks = ["Miembro 1":"Eddard", "Miembro 2":"Rob", "Miembro 3":"Catelyn", "Miembro 4":"Sansa"]
```

Como ves, en el primer ejemplo, hemos declarado un diccionario especificando su tipo [Int: String].

En el segundo ejemplo, hemos declarado un diccionario del mismo tipo, pero en este caso no lo hemos especificado, por lo que es el propio compilador quien lo infiere.

En el tercer ejemplo hemos creado un diccionario de tipo [String: String] sin especificar el tipo.

En estos ejemplos puedes observar que tanto para las claves como para los valores podemos utilizar cualquier tipo de dato.

Al igual que vimos con los arrays, puede que en algún momento queramos declarar un diccionario vacío y posteriormente añadirle elementos. Haríamos lo siguiente:

```
//Declaración de un Diccionario vacío
var topGames = [Int: String]()

//Inicialización del diccionario vacío
topGames = [3:"Rime", 2:"ICO", 1:"Zelda"]
```

Como ves, a la hora de declarar un diccionario vacío, si que es imprescindible especificar su tipo.

Una vez que lo hemos declarado podremos inicializarlo con elementos del tipo que hemos especificado en su declaración.

En este último ejemplo, hemos creado un diccionario vacío de tipo Int:String y lo hemos llenado con elementos de tipo Int:String.

Si quisiéramos acceder a cualquiera de los elementos almacenados, tendremos que utilizar su clave entre corchetes para poder acceder. Veamos como acceder al elemento del array *topGames*, almacenado en la clave 2:

topGames[2]	"ICO"
-------------	-------

Veamos, al igual que con los arrays, como añadir o borrar elementos de un diccionario y como mostrar su contenido por pantalla:

```
//Añadiendo un elemento al diccionario  
topGames[4] = "Monkey Island"  
  
//Eliminando un elemento del diccionario  
topGames.removeValue(forKey: 1)  
  
//Mostrando el contenido del diccionario  
print(topGames)
```

"Monkey Island"
"Zelda"
"[2: "ICO", 3: "Rime", 4: "Monkey Island"]\n"

En los ejemplos puedes observar que para añadir un elemento, simplemente especificamos el nombre del diccionario, seguido de la clave que queremos asignarle al elemento y en la parte derecha de la asignación dicho elemento.

Si queremos eliminar un elemento del diccionario, usamos el método `removeValue()` especificando la clave que tiene el elemento que queremos eliminar.

Para mostrar el contenido de un diccionario, al igual que con los arrays, usamos el método `print()`.

Para finalizar esta sección sobre diccionarios, nos quedaría ver un ejemplo en el que recorramos un diccionario y accedamos a los elementos del mismo. Aquí lo tienes:

```
101 //Recorriendo un diccionario  
102 for (ranking, videojuego) in topGames {  
103     print("El juego \(videojuego) ocupa la posición \(ranking) en el ranking")  
104 }
```

El juego ICO ocupa la posición 2 en el ranking
El juego Rime ocupa la posición 3 en el ranking
El juego Monkey Island ocupa la posición 4 en el ranking

Si te das cuenta, hemos utilizado las variables `ranking` y `videojuego` para almacenar respectivamente la clave y el valor de cada elemento. De esta forma podemos acceder a todo el contenido del diccionario.

SETS

WWW.EFECTOAPPLE.COM

Un set (también llamado conjunto) es una colección NO ORDENADA de datos.

En este punto seguramente te preguntarás, ¿entonces en qué se diferencian de los diccionarios?

Se diferencian en 2 puntos:

- No trabajan con parejas clave-valor
- No admiten elementos repetidos

Una vez sabido esto y teniendo en cuenta que de los 3 tipos de colecciones es el que menos se usa con diferencia, veamos las operaciones más habituales que suelen darse cuando trabajamos con sets:

Creación de objetos Set:

```
//Declaración de un Set de tipo Int especificando su tipo
let numerosLost = Set<Int>([4, 8, 15, 16, 23, 42])

//Declaración de un Set de tipo Int: String sin especificar su tipo
let edades = Set([4, 8, 15, 16, 23, 42])

//Declaración de un Set de tipo String

var starks = Set(["Eddard", "Rob", "Catelyn", "Sansa"])
```

Como ves, en el caso de los sets, debemos siempre añadir que estamos creando un objeto de tipo Set, en el momento de crearlo.

Añadir un elemento a un Set:

```
//Añadimos un elemento al set llamado starks
starks.insert("John Snow")
```

Eliminar un elemento de un Set:

```
//Eliminamos el elemento Rob del set starks  
starks.remove("Rob")
```

Para recorrer un Set, como cualquier otra colección, usaremos un bucle for-in, obteniendo cada uno de los datos del set en cada vuelta:

```
132 //Eliminamos el elemento Rob del set starks  
133 starks.remove("Rob")  
134  
135  
136 //Recorremos el set y mostramos por pantalla sus elementos  
137 for stark in starks {  
138     print("\(stark) es miembro de la familia Stark")  
139 }
```



```
Eddard es miembro de la familia Stark  
Sansa es miembro de la familia Stark  
John Snow es miembro de la familia Stark  
Catelyn es miembro de la familia Stark
```

TUPLAS

WWW.EFECTOAPPLE.COM

Podemos definir una tupla como una asociación rápida entre varios objetos.

Para especificar una tupla, los objetos que la forman deberán ir entre paréntesis. Esos paréntesis unen los objetos en un único elemento: La Tupla.

Además, hay que tener en cuenta que los objetos de una tupla tienen tipo propio y la tupla también.

Exactamente igual que las constantes y las variables, podemos definir tuplas especificando su tipo o sin especificar su tipo.

Veamos algunos ejemplos:

```
//Declaración de una tupla sin especificar el tipo  
var primeraTupla = (26, "Esta es la primera tupla", 14)  
  
//Declaración de una tupla especificando el tipo  
var segundaTupla : (Int, String, Int) = (62, "Esta es la segunda tupla", 41)
```

Tal y como hemos comentado, los objetos de estas dos tuplas tienen su tipo propio: El primer elemento es un Int, el segundo un String y el tercero un Int.

Además las tuplas tienen también su tipo. En este caso, ambas tuplas serán de tipo (Int, String, Int)

Fíjate lo que ocurrirá si declaramos la siguiente tupla:

```
//Declaración de una tupla especificando el tipo  
var terceraTupla : (Int, String, Int) = (26, "Esta es la segunda tupla", [3])
```

Recibimos un error de compilación en el tercer elemento de la tupla.

¿Por qué?

Muy sencillo. En la declaración de la tupla hemos especificado que el tercer elemento de la tupla será de tipo Int y en realidad le hemos pasado un array con un Int.

Por eso el compilador nos avisa que algo estamos haciendo mal.

Si declaramos una nueva tupla como esta:

```
let codigoError1 = (404, "Not Found")
```

Veremos como en la parte derecha de nuestro Playground se muestra lo siguiente:

```
(.0 404, .1 "Not Found")
```

¿Qué quieren decir estos ".0" y ".1"?

Simplemente indica la posición de cada uno de los elementos que conforman la tupla. El elemento 404 almacenado en la posición 0 y el elemento "Not Found" almacenado en la posición 1 de la tupla.

Existe la posibilidad de darles nombres a cada uno de los elementos de una tupla.

Aquí tienes un ejemplo:

```
let codigoError2 = (codigo: 404, description: "Not found")
```

En este caso, al segundo elemento le damos el nombre "description".

Y luego, podremos acceder a los elementos de la tupla a través de este nombre:

```
print(codigoError2.description)
```

Al escribir esta última linea, puedes ver que en la consola de tu Playground se muestra "Not found"

Existen dos tuplas especiales sobre las que tenemos que hablar.

La Tupla Vacía ()

La tupla () se utiliza para indicar la “nada”.

Una función que no devuelve nada, en el fondo si que está devolviendo algo, la tupla vacía: ()

Existe un typealias para ella llamado Void.

Ejemplo de declaración de una tupla vacía:

```
let tuplaVacia = ()
```

⚠ Constant 'tuplaVacia' inferred to have type '()', which may be unexpected

Como puedes ver, cuando hacemos esta declaración, al compilador le parece raro (lanza un warning) pero no pone más pegas.

La Tupla de un único elemento

La tupla de un elemento en realidad no existe, es el elemento en sí. Cuando intentamos crearla, el compilador lo que devuelve es el propio elemento.

```
let tuplaUnica = (9)
```

En realidad tuplaUnica simplemente es un 9.

Para finalizar faltaría mencionar que las tuplas solo pueden almacenar datos. No puedes almacenar funciones dentro de una tupla. Algo que si puedes hacer por ejemplo con las Clases y las Structs.

Se podría decir que son similares a las estructuras de C que utilizamos en Objective-C, como por ejemplo CGRect o CGPoint, solo agrupan valores y no tienen métodos asociados a ellas.

Lo importante es que recuerdes que deberás usar Tuplas cuando tengas que asociar dos objetos para algo muy sencillo.

PATTERN MATCHING

WWW.EFECTOAPPLE.COM

El **Pattern Matching** es una de las características de Swift que más nos facilitan la vida a los desarrolladores.

Simplificando mucho se puede decir que el **Pattern Matching** es una asignación de variables avanzada, donde nosotros le damos alguna pista al compilador y él solo es capaz de averiguar a qué nos estamos refiriendo.

Dicho así no queda nada claro, así que vamos a verlo con un ejemplo:

```
//Declaramos una tupla de 2 elementos
let simpsons = ("Homer", "Lisa")

//A través de Pattern Matching extraemos los dos componentes y los guardamos en dos
//variables diferentes
var (padre, hija) = simpsons
```

Swift es capaz de hacer él solo el siguiente razonamiento:

"Ha declarado una tupla de 2 elementos y ahora me dice que asigne esa tupla a dos variables diferentes. Así que creo que lo que quiere decir, es que asigne el elemento 1 de la tupla a la variable 1 (padre) y el elemento 2 de la tupla a la variable 2 (hija)"

Si después de estas líneas escribes:

```
print(padre)
```

Verás como se muestra por pantalla "Homer". Por lo que el Pattern Matching ha funcionado correctamente.

En ocasiones puede que no nos interese almacenar todos los elementos de una tupla, si no solo alguno en concreto.

En este caso utilizaremos la **Variable Anónima**, que se referencia con un guión bajo: _

Veámoslo con otro ejemplo:

Imaginemos que solo nos interesa el segundo elemento de la tupla: **Lisa**, el primer elemento: **Homer** nos da igual.

Podríamos repetir la operación realizada antes, crear dos variables: padre, hija y olvidarnos de la variable padre, dejarla morir.

Es una forma de trabajar, aunque no la más óptima.

Existe un mecanismo más sencillo que nos ahorra tener que crear variables que no vamos a usar y es utilizar la variable anónima (_).

Con la variable anónima dejo claro que no necesito el valor del primer componente de la tupla y almaceno el segundo elemento en la variable hija.

```
var(_, hija) = simpsons
```

En este caso, en la variable hija se almacenaría el String "Lisa".

Si quieras comprobar si el Pattern Matching ha funcionado, puedes hacer la misma operación que hicimos antes y escribir:

```
print(hija)
```

Puedes ver como en pantalla se muestra el String "Lisa".

FUNCIONES

WWW.EFECTOAPPLE.COM

¿Recuerdas el concepto de Inferencia de Tipos?

¿Sí?

Pues lamentablemente no está disponible para las funciones.

Swift todavía no es capaz de inferir el tipo de los parámetros y el retorno, por lo que tenemos que especificarlos.

Esto es bastante engorroso.

¿Cómo podemos solucionar en parte este problema?

Usando typealias.

Las funciones pueden devolver cualquier tipo. Incluyendo tuplas.

Cuando se dice que las funciones en Swift pueden devolver varias cosas, en realidad a lo que se refieren es que pueden devolver tuplas. Y como sabemos una tupla es una combinación de varios elementos. Después podemos utilizar el Pattern Matching para obtener esos elementos de las tuplas.

Las funciones pueden aceptar otras funciones como parámetros. Al igual que ocurre en JavaScript. Además las funciones pueden devolver otras funciones.

Las funciones en Swift, al igual que los bloques en Objective-C capturan el entorno léxico.

En cuanto a los nombres de los parámetros, hay que decir, que las funciones pueden tener nombres internos y externos.

Puedes tener una función que tenga un parámetro y ese parámetro tiene un nombre que solo es visible dentro de la función y tiene un nombre distinto que se utiliza desde fuera.

¿Para que sirve esto si lo único que hace es complicarnos la vida?

Esto se añadió para intentar replicar la sintaxis de los métodos en Objective-C.

Aquí tenemos un ejemplo de una función sencilla:

```
func producto(operador1: Int, operador2: Int) -> Int{  
    return operador1 * operador2  
}
```

Esta es la información mínima que tenemos que escribir para crear una función.

En este ejemplo, vemos que los parámetros únicamente tienen un nombre cada uno. En este caso, podremos utilizar este nombre, tanto dentro de la función, como fuera (Cuando realicemos la llamada a la función).

Si quisiéramos llamar a esta función, haríamos esto:

```
producto(10, operador2: 2)
```

Como puedes ver, en la llamada, no especificamos el nombre del primer parámetro, simplemente pasamos su valor = 10. En cambio en el segundo parámetro si que especificamos su nombre = operador2. Esto se debe a lo siguiente:

En Swift, cuando llamas a una función, el nombre del primer parámetro se omite

En este otro ejemplo, vemos como creariamos un nombre externo para uno de los parámetros:

```
func operacion(operador1:Int, operador2:Int, ultimoOperador operador3:Int) -> Int{  
    return (operador1 + operador2) * operador3  
}
```

Como puedes ver, el último parámetro tiene 2 nombres:

- Nombre interno: operador3
- Nombre externo: ultimoOperador

Esto nos permite simular un poco la sintaxis de los métodos de Objective-C. Desde dentro de la función haríamos referencia a los parámetros como: operador1, operador2 y operador3. Cuando tuviéramos que llamar a esta función haríamos:

```
operacion(10, operador2: 5, ultimoOperador: 2)
```

Como puedes ver, para el último parámetro utilizamos su nombre externo. Esto es obligatorio, ya que:

Si defines un nombre externo para el parámetro de una función, debes utilizar su nombre externo cuando realices la llamada a esa función

Existe una forma de poder omitir el nombre de un parámetro, para ello tienes que utilizar el carácter `_`

Aquí tienes un ejemplo:

```
/* Declaramos una función, especificando con _ que queremos
   omitir el nombre del tercer parámetro */
func threeProducts(num1: Int, num2: Int, _ num3: Int) -> Int
{
    return num1 * num2 * num3
}

/* Por lo que cuando hagamos la llamada a esta función
   no será necesario que especifiquemos el nombre del
   tercer parámetro */
threeProducts(2, num2: 4, 6)
```

Cómo puedes ver, haríamos la llamada sin especificar ni el nombre interno ni externo del parámetro 3.

Como es posible que a estas alturas tengas un buen cacao con los nombres de los parámetros (No te preocupes, yo estaba igual que tu la primera vez que vi todo esto), aquí tienes un pequeño resumen sobre los nombres de los parámetros:

- Cuando llamamos a una función, el nombre del primer parámetro siempre se omite
- El resto de parámetros tendrán el mismo nombre interno y externo a no ser que especifiques un nombre externo
- Cuando especifiques un nombre externo a un parámetro, deberás utilizar para llamar a la función siempre ese nombre externo

Para acabar de complicar todo esto, hay que decir que las los parámetros de las funciones pueden tener valores por defecto.

Por ejemplo:

```
func addLastName(name:String, lastName: String = " Simpson") -> String{  
    return name + lastName  
}
```

Como puedes ver, en esta función, el parámetro lastName tiene como valor por defecto "Simpson".

Si hacemos la siguiente llamada:

```
addLastName("Homer")
```

Nuestra función devolverá "Homer Simpson"

Pero si hacemos la siguiente llamada, especificando el segundo parámetro:

```
addLastName("Ned", lastName: " Flanders")
```

Nuestra función devolverá "Ned Flanders"

Una de las cosas que hemos comentado anteriormente es que una función puede devolver una tupla. Aquí podemos ver un ejemplo:

```
func famousFamily(type:Int) -> (String, String, String){  
    var familyMembers : (String, String, String)  
  
    switch type{  
        case 1:  
            familyMembers = ("Homer", "Marge", "Lisa")  
        case 2:  
            familyMembers = ("Eddard", "Sansa", "Catelyn")  
        default:  
            familyMembers = ("Goku", "Chichi", "Pan")  
    }  
  
    return familyMembers  
}
```

NOTA: En Swift, los switches siempre tienen que tener un valor por defecto (default) para en el caso de que el switch reciba un valor no contemplado, pueda devolver algo. Es obligatorio.

En este caso se trata de una tupla que almacena 3 Strings, pero podría ser una tupla que almacenara 2 Strings y 1 Int.

Después podríamos utilizar PATTERN MATCHING para llamar a esta función y quedarnos únicamente con los 2 últimos elementos de la tupla.

Por ejemplo, imagina que únicamente queremos guardar los miembros del sexo femenino de la familia Simpson. Haríamos esto:

```
var (_, girl1, girl2) = famousFamily(1)
```

Además de todo lo que hemos visto, hay que decir, que una función puede pasarse como parámetro de otra.

En estos casos, para evitar complicaciones lo mejor es ayudarnos de los typealias.

Lo puedes ver en este ejemplo:

```
//Definimos un typealias para que el código sea mas legible
typealias IntToIntFunc = (Int) -> Int

//Definimos 3 funciones diferentes

func maths(f:IntToIntFunc, n:Int) -> Int{
    return f(n)
}

func half(a:Int) -> Int {
    return a/2
}

func subtract20(a:Int) -> Int{
    return a - 20
}

//Llamamos a la función maths pasándole 2 funciones diferentes

maths(half, n:26)
maths(subtract20, n:46)
```

En este ejemplo puedes ver que cuando pasamos una función como parámetro de otra, hay que especificar el tipo de esa función.

En nuestro caso el tipo de la función es (Int) -> Int.

Lo que hemos hecho para simplificar la sintaxis es crear el typealias IntToIntFunc.

De esta forma especificamos el tipo de la función que pasamos como parámetro a la función maths.

Como puedes ver al final del ejemplo, hacemos la llamada a la función maths, pasándole la función half y posteriormente pasándole la función subtract20.

Como resultados, a estas dos llamadas, en nuestro Playground obtenemos:

13
26

Viendo este último ejemplo de funciones,

¿no te recuerdan a algún concepto de Objective-C?

Exacto. A los bloques.

Las funciones en Swift pueden funcionar como los bloques en Objective-C

Si ves que todo esto es algo complejo, no te preocupes, todo se debe a la sintaxis. Es necesario un tiempo de adaptación hasta que seas capaz de leer rápidamente el código en Swift.

Además, para tu alivio, te adelanto que pasar funciones a otras funciones y devolver funciones no será algo que tendrás que hacer todos los días.

Simplemente es conveniente que sepas que en Swift pueden hacerse cosas de este tipo.

Para acabar, hay que mencionar que también existen funciones que devuelven funciones y funciones dentro de funciones.

Las funciones dentro de funciones, son algo propio de Swift muy útil, ya que si estamos trabajando y vemos que la función que estamos desarrollando se está complicando demasiado, podremos crear funciones dentro de nuestra función que realicen trabajos concretos de la función madre.

También se pueden almacenar funciones dentro de un array. Ya que las funciones son un tipo más del sistema.

No veremos ejemplos de funciones dentro de funciones, funciones que devuelven otras funciones ni funciones almacenadas en arrays, porque considero que lo visto hasta ahora es más que suficiente para comenzar a trabajar con Swift.

Además vamos a manejar estos conceptos avanzados en el siguiente apartado, que son muy parecidos a los que acabamos de ver con las funciones

CLOSURES

WWW.EFECTOAPPLE.COM

Las Closures también conocidas como Clausuras, son FUNCIONES DE PRIMER NIVEL.

Se tratan también de funciones pero con algunas características avanzadas. Digamos que son SuperFunciones.

Lo único que cambian con respecto a las funciones que hemos visto en el apartado anterior es su sintaxis, ya que todas las Closures son funciones. A esta nueva sintaxis se le llama Sintaxis de Clausura.

RECUERDA
TODAS LAS CLOSURES SON FUNCIONES
ESTO ES SOLO SINTAXIS NUEVA

Hay que tener en cuenta que la inferencia de tipos funciona mucho mejor con la sintaxis de clausura que con la sintaxis simple.

Además, lo normal es que la sintaxis de clausura sea más limpia que la sintaxis simple.

Vamos a ver aspectos nuevos de las funciones que ya conocíamos. Simplemente se le da otro nombre, pero en el fondo es todo lo mismo.

Una Closure es lo que en Objective-C llamábamos Bloques. Al igual que éstos, captura el entorno léxico.

La sintaxis de una closure sería la siguiente:

```
{ (params) -> returnType in  
    body  
}
```

Lo que marca que se trata de una Closure son las llaves {}.

Al igual que las funciones, tendrá unos parámetros entre paréntesis y un tipo de retorno.

A partir de la palabra reservada *in* comenzaría el cuerpo de la Closure.

Aquí tienes un ejemplo de función simple:

```
func sum(a:Int) -> Int{  
    return a + 1  
}
```

Que equivaldría exactamente a esto, con la sintaxis de clausura:

```
let sum = {(a:Int) -> Int in
            return a + 42
        }
```

A partir del *in* es donde vendría el cuerpo de la Closure. Ahí es donde realmente hacemos algo.

Como puedes ver, en la sintaxis simple, lo que hacemos es declarar una función **sum** que le suma 1 al parámetro que le pasemos.

Esto convertido a sintaxis de clausura, se podría hacer declarando una constante **sum** donde almacenaremos directamente una closure.

¿Por qué es necesario crear una constante?

Esto se debe, a que en Swift, una linea de código no puede comenzar con una closure. Por lo que tendremos que declarar una variable o una constante donde almacenarla.

Imagina que ahora hacemos esto:

```
sum(3)
```

¿Qué función crees que se ejecutaría? ¿La función o la closure?

Se ejecutaría la closure. Porque...

Si tenemos dos funciones iguales y hacemos una llamada a dichas funciones, siempre tiene preferencia la que tiene la sintaxis de Closure.

Veamos, en este otro ejemplo, como podemos declarar un array donde almacenemos closures.

En este array vamos a almacenar 4 closures diferentes y cada una la vamos a añadir con una sintaxis distinta:

```
var closuresArray = [{(a: Int) -> Int in return a - 20},  
                     {a in return a + 26},  
                     {a in a * 14},  
                     {$0 * 8}]
```

En el momento en el que agregamos la primera closure al array, se convierte en un array de funciones/closures que reciben un Int como parámetro y devuelven un Int. Podremos añadir tantas funciones o closures a este array como queramos siempre que sean de ese tipo: (Int)-> Int

Vamos a ver como hemos añadido las 4 closures:

Función 1: {(a: Int) -> Int in return a -20}

La hemos añadido utilizando su sintaxis completa.

Función 2: {a in return a + 26}

En este caso, hemos añadido la closure sin especificar el tipo del parámetro ni el tipo del valor de retorno. El tipo de la Closure **(Int)-> Int** se infiere, ya que la closure que hemos añadido anteriormente es de este tipo.

Función 3: {a in a * 14}

Si nuestra closure solo tiene una expresión, el tipo del valor de retorno es implícito

Función 4: {\$0 * 8}

Esta última closure tal vez te extrañe un poco. Simplemente es un ejemplo para dejar claro, que se puede acceder a los parámetros de una closure por su posición. En este caso \$0, sería el parámetro situado en la posición 1 (El único que tiene).

Conclusión

Las closures, al igual que las funciones, son un tipo más, por lo que no hay problema para guardarlas en colecciones.

No solo esto. Una vez que tenemos nuestras closures almacenadas en closuresArray, podemos acceder a cualquiera de ellas y llamarlas:

```
closuresArray[2](2)
```

En este caso, accedemos a la closure almacenada en la posición 2 y la ejecutamos pasándole un 2.

Como puedes ver en tu playground, funciona correctamente, obteniendo un resultado de 28.

OPCIONALES

WWW.EFECTOAPPLE.COM

Primero, para tratar de explicar los Opcionales, me gustaría hacerte una pregunta:

¿Cómo representar y manejar un valor que no existe?

En muchos lenguajes, entre ellos Objective-C se utilizaba nil para representar que un valor no existe.

¿Y que es lo que pasaba en Objective-C cuando tratábamos de utilizar una variable no inicializada? Es decir, inicializada a nil.

No pasaba nada, el sistema lo ignoraba.

En algunos lenguajes si tratábamos de acceder a una variable no inicializada, podía suceder que nuestra aplicación sufriera un crash. Esto sucede por ejemplo en C.

En otros, esta situación lo que producía era una Excepción. Si no tratábamos correctamente esa excepción, nuestra aplicación también podía caerse. Por ejemplo en Java.

En Swift ocurre exactamente lo mismo que en Java, un valor nil puede hacer que nuestra aplicación sufra un crash.

Veamos ahora ejemplos que nos harán ver que es lo que pasa en Swift cuando trabajamos con variables no inicializadas.

```
//Declaramos una variable x y no le asignamos ningún valor, la dejamos sin inicializar
var x: Int

//Después intentamos trabajar con esa variable
var y = x + 1
```

! Variable 'x' used before being initialized

Como puedes ver, si declaramos una variable y no la inicializamos, en el momento en que queramos utilizarla, obtendremos el siguiente mensaje de error: **variable 'x' used before being initialized**.

Lo mas interesante de este ejemplo es que este error lo está reportando el compilador. Se está produciendo en tiempo de compilación en lugar de en tiempo de ejecución. Si el compilador no nos avisara de esto, el error sucedería al ejecutar nuestra aplicación y se produciría un crash.

Swift nos informa en tiempo de compilación para evitar que utilicemos de forma accidental, variables que no han sido inicializadas.

Esto sucede con las variables, pero, ¿ocurre lo mismo con las constantes? Veámoslo con otro ejemplo:

```
//Declaramos una constante z
let z: Int

//Como vemos, Swift nos permite declarar una constante y dejarla
//sin inicializar. No obtenemos ningún error en tiempo de compilación.

//También nos permite inicializarla más adelante
z = 10

//¿Pero qué ocurre si intentamos modificar su valor?
z = 20
```

Immutable value 'z' may only be initialized once

Como puedes ver, Swift nos permite dejar una constante sin inicializar. Incluso nos permite inicializarla más adelante. Lo que si deja claro el compilador con este error, es que solo nos permite inicializar las constantes una única vez.

Si volvemos al primer ejemplo que hemos visto, podría parecer que Swift no nos permite trabajar con una variable que no tenga ningún valor, es decir, que sea nil. Pero no es así.

Existe una forma de hacer que una variable quede sin inicializar y por tanto almacene nil y que además podamos trabajar con ella. Para poder hacer esto tendremos que utilizar los **OPCIONALES**.

Para ello, a la hora de declarar una variable tendremos que **indicar de forma explícita** que esa variable si no le asignamos ningún valor, almacenará nil. Para poder indicar esto, colocaremos el carácter ? justo después del tipo de la variable. Aquí tienes un ejemplo:

```
//Declaramos una variable opcional (Puede almacenar nil)
var a: Int?

//Como puedes ver a la derecha del playground, nuestra variable a almacena nil.
//A este tipo de variable se le consideraría una variable de tipo opcional.

//Podemos asignarle otro valor:
a = 1

//E inmediatamente después si queremos que vuelva a almacenar nil se lo podemos asignar:
a = nil
```

nil

1

nil

Por tanto, la diferencia de Swift con otros lenguajes, es que si queremos que una variable pueda almacenar el valor nil tendremos que especificarlo de forma explícita al declarar esa variable, es decir, crearla como un OPCIONAL.

Un opcional es simplemente una variable que puede contener nil.

También es importante que tengas en cuenta esto:

UNA CONSTANTE NO PUEDE SER OPCIONAL

Por tanto, únicamente trabajaremos con opcionales cuando se traten de variables.

Ahora vamos a ver un ejemplo que tal vez te sorprenda un poco:

```
//Declaramos un Int opcional y le asignamos el valor 1
var f: Int? = 1

//Le sumamos 9 a la variable f
f + 9
```

Value of optional type 'Int?' not unwrapped; did you mean to use '!' or '?'

Hemos hecho algo tan simple como declarar un opcional de tipo Int, le hemos asignado el valor 1 y al intentar sumarle 9 a esa variable obtenemos un error de compilación.

¿Por qué?

Si nuestra variable f vale 1 y es de tipo Int, ¿Por qué no podemos sumarle 9?

La explicación a esto se debe a la propia definición de los Opcionales.

Un opcional es como una caja que envuelve nuestra variable. Esta caja puede contener el valor de la variable o puede contener nil. Puede que nosotros no lo sepamos hasta que abramos esta caja. Para acceder al valor de la variable tendremos que abrir esta caja. Para abrir esta caja tendremos que utilizar el carácter !

Esta es la sintaxis que tendremos que utilizar cuando queramos acceder al valor almacenado dentro de un opcional.

De esta forma, para solucionar el último ejemplo, únicamente añadimos ! a nuestra variable y vemos como en la parte derecha del playground obtenemos de resultado 10:

```
//Le sumamos 9 a la variable f  
f! + 9
```

10

A modo de resumen es importante que te quedes con esto:

Para declarar una variable como opcional (y de esta manera permitir que almacene nil) utilizaremos ?
Para acceder al valor de un opcional utilizaremos !

El problema de utilizar ! para acceder al valor de un opcional es que se trata de algo poco seguro.

¿Por qué decimos esto?

Fíjate en el siguiente ejemplo:

```
//Hacemos que f sea nil  
f = nil  
  
//Creamos una nueva variable y le asignamos el valor contenido en f, es decir, nil  
var nuevaVariable = f!  
⚠ Execution was interrupted, reason: EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0).
```

Lo que ocurre cuando hacemos esto es que obtenemos un ERROR EN TIEMPO DE EJECUCIÓN, no en tiempo de compilación como sucedía en los anteriores errores. Es decir, produce que nuestra aplicación se cierre.

Por tanto, si utilizamos los opcionales para asignarle el valor de una variable a otro objeto y resulta que ese valor es nil, obtendremos un error en tiempo de ejecución y nuestra aplicación fallará.

¿Cuál es la forma segura entonces de poder realizar una asignación de un opcional a otra variable?

Utilizando la asignación opcional

La asignación opcional simplemente consiste en comprobar primero si nuestro opcional tiene algún valor y en el caso de que no sea nil, realizar la asignación.

Si por ejemplo, quisiéramos trabajar con la variable f de forma segura, haríamos esto:

```
/* Realizamos la asignación opcional de nuestra variable f
a una constante value */
if let value = f {
    //Hacemos lo que queramos con la constante value
}
```

Como puedes ver, lo primero que hacemos es comprobar si nuestro opcional tiene algún valor y en el caso de que no sea nil se lo asigna a la constante value. Luego ya trabajamos con value, que será realmente un Int y no un opcional Int, como era f.

Por tanto, para realizar asignaciones seguras de un opcional a otra variable/ constante tendremos que utilizar la asignación opcional.

A modo de resumen, recuerda:

Para acceder a un opcional tienes dos opciones:

- Acceder de forma segura con la asignación opcional
- Acceder de forma insegura con !

La práctica recomendada es que utilices siempre que puedas la forma segura, llamada **Asignación Opcional** y evites la forma insegura o también llamada **Desempaquetamiento Forzado ó Inseguro**.

A pesar de que siempre se recomienda la asignación opcional, tiene un problema y es que si trabajamos con varios opcionales al mismo tiempo, puede generar código poco legible.

Vamos a verlo con un ejemplo:

```
//Declaramos un opcional string llamado cadenaOpcional1  
var cadenaOpcional1: String? = "Este es el primer String"  
  
//Declaramos otro opcional string llamado cadenaOpcional2  
var cadenaOpcional2: String? = "Este es el segundo String"  
  
//Como tenemos que trabajar con ambos strings y vamos a utilizar asignación opcional,  
//tendremos que utilizar 2 if anidados:  
if let string1 = cadenaOpcional1 {  
    if let string2 = cadenaOpcional2 {  
        print("La cadena1 es \(string1), la cadena 2 es \(string2)")  
    }  
}
```

Ahora imagina que en lugar de con 2 opcionales tenemos que trabajar con 6. Lo que sucederá es que tendremos bloques de if anidados excesivamente grandes que generarán código poco legible.

Para solucionarlo, existe otra sintaxis que sustituye a los dos if anidados anteriores:

```
if let string1 = cadenaOpcional1, string2 = cadenaOpcional2 {  
    print("La cadena1 es \(string1), la cadena 2 es \(string2)")  
}
```

Como puedes ver se trata de una sintaxis mucho más limpia que facilita la lectura del código.

Para finalizar este apartado, estas son las recomendaciones para el trabajo con Opcionales:

- Trata de no declarar variables como opcionales.
- Solo utiliza opcionales cuando realmente necesites gestionar el caso de que algo puede ser nil.
- Evita forzar la obtención del valor real con !, mejor utiliza la asignación opcional.

CLASES

WWW.EFECTOAPPLE.COM

Las clases en Swift son muy similares a las clases en Objective-C, aunque tienen algunas particularidades.

Al igual que en Objective-C, una clase tiene sus propiedades, funciones e inicializadores. Los inicializadores son algo más complejos que en Objective-C.

Las clases en Swift, tienen herencia única. Además, Apple nos permite hacer que una clase de Swift herede de una de Objective-C y viceversa.

Una de las particularidades de Swift, es que cuando creas una clase, no tienes por qué asignarle una superclase. Es decir, no es necesario que nuestra clase herede de otra. Si que debes tener en cuenta, que si envías el mensaje *super* a una clase sin padre, tu código ni siquiera va a compilar.

Además las clases en Swift pueden tener atributos de clase (estáticos).

Las clases se guardan en el montículo (Heap) y su memoria la gestiona ARC.

Vamos a comenzar con la parte práctica creando una clase llamada Persona. Esta clase tendrá 3 propiedades:

- Una constante que será el **nombre**
- Una variable que será la **altura**
- Una variable que será el **peso**

Ahora ve a tu playground y crea esta clase:

```
class Persona {  
    let nombre = "Homer"  
  
    var altura: Double  
    var peso: Double  
}
```

Como puedes ver, la declaración de nuestra clase Persona produce un error de compilación:

Class 'Persona' has no initializers

Este error se debe a que en las clases de Swift, es necesario inicializar las propiedades que declaramos. En este caso, la propiedad nombre está inicializada a pero, altura y peso no.

Vamos a solucionarlo:

```
class Persona {  
    let nombre = "Homer"  
  
    var altura: Double = 170  
    var peso: Double = 108  
}
```

Comprobamos en la wikipedia cuando mide y pesa Homer Simpson, inicializamos ambas propiedades y el error desaparece.

La propiedad nombre de nuestra clase será readonly, es decir, de solo lectura, ya que la hemos declarado como una constante.

Si te fijas bien, puedes ver que se trata de una clase base, es decir, no hereda de ninguna otra clase, ya que detrás del nombre de la clase **Persona**, no aparece su clase padre.

Aunque en la declaración de la clase no aparezca su clase padre, no es del todo verdad que nuestra clase Persona no hereda de ninguna clase.

A pesar de que nosotros no lo vemos si que hereda de una clase, digamos, "invisible" llamada **SwiftObject** y que sería como el NSObject de Objective-C.

Como cualquier clase, además de atributos, podemos tener funciones.

Añadamos a la clase Persona una función que nos devuelva el peso de la persona:

```
class Persona {  
    let nombre = "Homer"  
  
    var altura: Double = 170  
    var peso: Double = 108  
  
    func getPeso() -> Double{  
        return peso  
    }  
}
```

De esta forma, tendremos una clase Persona, que tendrá 2 atributos: **altura** y **peso** y una función que nos devolverá el peso de la persona: **getPeso()**.

Una vez declarada nuestra clase si queremos crear un objeto de tipo Persona. Para ello haremos lo siguiente:

```
let persona = Persona()
```

Una vez que tengamos nuestro objeto persona, podremos llamar al método **getPeso()**:

```
var pesoHomer = persona.getPeso()
```

En una clase de Swift, además podemos definir Propiedades de Tipo.

¿Qué son las propiedades de tipo?

Son lo que se conoce en otros lenguajes como Propiedades de Clase.

Las Propiedades de Tipo te permiten asociar propiedades a un tipo concreto, de forma que podrás acceder a ellas directamente a través de su tipo, en lugar de tener que declarar una instancia de ese tipo antes.

Las Propiedades de Tipo pueden ser tanto constantes como variables.

De la misma forma, nos permite crear Funciones de Tipo.

Para crear propiedades y funciones de tipo utilizaremos la palabra reservada: **static**.

Veámoslo continuando con nuestro ejemplo:

```
class Persona {  
    static let pesoDefault = 75  
    var altura: Double = 170  
    var peso: Double = 108  
    func getPeso() -> Double {  
        return peso  
    }  
    static func returnPesoDefault() -> Int{  
        return pesoDefault  
    }  
}
```

En este caso hemos añadido la propiedad de tipo **pesoDefault** y la función de tipo **returnPesoDefault**.

Para acceder a ellas utilizaremos directamente el Tipo, sin necesidad de declarar ninguna instancia antes:

```
Persona.pesoDefault  
Persona.returnPesoDefault()
```

Existen un tipo de propiedades especiales conocidas como **Propiedades Computadas**.

Las propiedades computadas te permiten generar valores dentro de las propiedades.

Si has trabajado con Objective-C recordarás que existía la posibilidad de sobreescribir los métodos accesores (Getter y Setter) de las propiedades de una clase.

Como en Swift no podemos hacer eso, tenemos las propiedades computadas para poder realizar esa función.

Lo vas a ver muy claro en el siguiente ejemplo.

Vamos a crear una clase llamada Personaje que tendrá una propiedad computada llamada fullName y una propiedad normal y corriente llamada father.

Lo que queremos es que si el objeto de tipo Personaje tiene padre, su nombre completo sea por ejemplo:

“Jaime Lannister hijo de Tywin Lannister”

Pero si el objeto de tipo Personaje no tiene padre, su nombre simplemente será:

“Jaime Lannister”

Para ello tendremos que sobreescibir el método get de la propiedad fullName utilizando una propiedad computada:

```
class Personaje {  
  
    //Propiedad name  
    var name: String?  
    //Propiedad father  
    var father: String?  
  
    //Propiedad Computada fullName  
    var fullName: String {  
        //Sobreescribimos el método get  
        get{  
            var n = ""  
            if let theName = name{  
                n = theName  
            }  
            if let theFather = father{  
                n = n + " hijo de \(theFather)"  
            }  
  
            return n  
        }  
    }  
}
```

Si además del método set, quisiéramos sobreescibir el método set, dentro de la propiedad fullName, a continuación del get{}, escribiríamos otro bloque de código que fuera set{}.

Otro concepto interesante dentro de las clases son los **Observadores de Propiedades**.

Si has programado en Objective-C tal vez conozcas KVO. Los Observadores de Propiedades son un sistema similar a KVO.

Si no has programado antes en Objective-C o no conoces KVO no te preocupes porque vamos a explicar detalladamente en que consiste este concepto.

Los observadores de propiedades te permiten enterarte siempre que algo quiera modificar el valor de una propiedad.

Su sintaxis es muy similar a la de las Propiedades Computadas. De hecho, suelen declararse una detrás de otra, primero la propiedad computada y después el observador de dicha propiedad.

El observador dispone de dos métodos: *willSet* y *didSet*.

WillSet nos avisa cuando el valor de la propiedad va a cambiar.

DidSet nos avisa cuando el valor de la propiedad ya ha cambiado.

Además el willSet te permite pasar como parámetro el nuevo valor de la propiedad y el didSet te permite pasar como parámetro el viejo valor de la propiedad. Por si te interesa compararlos.

En este ejemplo puedes ver la declaración de un observador de la propiedad father:

```
class Personaje {  
  
    var fullName: String = ""  
    var father: String = ""{  
        willSet{  
            print("El valor de la propiedad father va a cambiar")  
        }  
        didSet{  
            print("El valor de la propiedad father ha cambiado")  
        }  
    }  
}
```

STRUCTS

WWW.EFECTOAPPLE.COM

Una struct es simplemente una estructura de datos.

Swift las usa muchísimo. Para que te hagas una idea, Array y Dictionary no son clases, sino structs.

A modo de resumen, estas son algunas de las características de las Structs:

- Son muy similares a las clases
- No se pueden extender mediante herencia
- Son tipos de datos que se pasan por valor y no por referencia como las clases
- Generalmente no se modifican. Suelen ser inmutables

La sintaxis para la creación de un struct sería la siguiente:

```
struct Example_Struct {  
    //Propiedades que declaremos en la struct  
} //Funciones que declaremos en la struct
```

Aquí tienes un ejemplo de como crear una struct:

```
struct Character {  
    var name : String  
    var lastName: String  
    var age: Int  
    var family: String  
  
    func getName() -> String {  
        return name  
    }  
}
```

Como ves, en este ejemplo, la struct es exactamente igual a la declaración de una clase. Tiene una serie de propiedades y una función que definen la struct.

Si después queremos crear un objeto del tipo de esta struct, haremos esto:

```
let Rhaegar = Character(name: "Rhaegar", lastName: "Targaryen", age: 24, family: "Targaryen")
```

Si te das cuenta, para crear un objeto de esta struct no hemos tenido que crear ningún inicializador. Esto se debe a que todos los structs tienen un **inicializador por defecto**, que nos permite pasarle los valores de cada variable.

Además de este inicializador por defecto, nosotros podemos crear tantos inicializadores como queramos, eso sí, todos principales o designados. Los structs no nos permiten crear inicializadores de conveniencia.

Vamos a añadirle un par de inicializadores a nuestro struct:

```
struct Character {  
    var name : String  
    var lastName: String  
    var age: Int  
    var family: String  
  
    init(name:String, lastName:String, age:Int, family:String) {  
        self.name = name  
        self.lastName = lastName  
        self.age = age  
        self.family = family  
    }  
  
    init(name:String, lastName:String) {  
        self.init(name:name, lastName:lastName, age:0, family:"")  
    }  
  
    func getName()-> String {  
        return name  
    }  
}
```

Hemos creado un inicializador con el que inicializar todos los parámetros y un inicializador que lo que hace es que llama al inicializador anterior pasándole valores vacíos para las propiedades **age** y **family**.

Puedes crear tantos inicializadores como quieras y puedes llamar de unos a otros utilizando **self.init**.

Además debes saber, que en el momento en que creas un inicializador en tu struct, el inicializador por defecto desaparece, por lo que tendríamos que crearlo nosotros por código si queremos volver a usarlo.

Estás viendo que los structs son muy similares a las clases de cualquier lenguaje de programación. Sin embargo además de que no permiten la herencia, tienen otra gran diferencia con respecto a las clases.

Se comportan de un modo diferente en memoria.

Las clases se guardan en el montículo (Heap), mientras que las structs se guardan en la Stack (Pila).

El hecho de que los structs estén en la pila tiene gran importancia. Hace que no pueda haber dos referencias al mismo objeto, como si ocurre con las clases. Por tanto, cuando queramos trabajar con structs, en lugar de pasárselas por referencia, tendremos que pasárselas por valor.

Cuando trabajamos con clases trabajamos con referencias, en lugar de manejar objetos directamente, utilizamos direcciones de memoria donde están contenidos dichos objetos.

Como esto no es posible con las structs, trabajamos directamente con los objetos. Es decir, pasamos las variables por valor. Cuando asignamos una variable de un struct a otro struct, en realidad estamos haciendo una copia idéntica de ese objeto, con el cual trabajaremos a partir de ahora.

Se trata de una gran diferencia, que debes conocer entre clases y structs.

Como hemos comentado antes, las structs suelen ser inmutables. Por tanto, los métodos de un Struct no deben modificar las propiedades del Struct.

Esa es la teoría.

Sin embargo, Swift nos ofrece un mecanismo para modificar las propiedades de un Struct.

Para poder modificar una propiedad de un struct, tenemos que marcar la función que vaya a modificar la propiedad, con la palabra reservada **mutating**.

Lo vemos con un ejemplo:

```
struct Character {
    var name = "Tyrion Lannister"

    //Marcamos la función como mutating para que pueda cambiar la propiedad name
    mutating func changeName(){
        name = "El Gomo"
    }
}
```

En este ejemplo, hemos definido la función **changeName** como mutating para que pueda modificar la propiedad **name** de la struct. Además también puedes ver que hemos inicializado la propiedad **name** directamente en su declaración.

Al ver que las structs son tan parecidas a las clases, tal vez te estés preguntando cuando debemos usarlas.

Como normal general, utilizaremos las structs cuando no nos interese que pueda modificarse el valor de un objeto desde diferentes sitios.

Como por ejemplo, si estamos trabajando en un entorno multihilo.

Las structs se utilizan mucho para enviar información de una cola a otra cuando utilizamos Grand Central Dispatch.

ENUMS

WWW.EFECTOAPPLE.COM

El siguiente concepto que vamos a ver son las **Enums**. Se conocen también como enumeraciones.

Se trata de otro elemento de Swift que puede sernos de gran utilidad.

Las enums se pueden definir como un conjunto de datos de un mismo tipo, que agrupa valores relacionados entre si.

Al revés que en Objective-C donde en las enum únicamente podían almacenar números, en Swift pueden almacenar números y también cadenas.

Vamos a verlo con un ejemplo.

Imagina que estamos trabajando en una aplicación que muestra información sobre el Sistema Solar.

Si en esta app quisiéramos representar los planetas de dicho sistema, tenemos claro, que un planeta únicamente puede ser uno de los 8 planetas que forman nuestro Sistema Solar.

Por tanto, podríamos declarar una enum llamada planet que almacenara este conjunto de valores:

```
enum planet{
    case mercurio, venus, tierra, marte, jupiter, saturno, urano, neptuno
}
```

De esta forma, cada vez que utilicemos la variable planet, podremos especificar uno de los valores de nuestra enum, utilizando un punto y asignándoselo a cualquier variable que creemos.

Podemos hacerlo de esta manera:

```
var closestPlanetToSun = planet.mercurio
```

O de esta otra:

```
var closetPlanetToSun:planet = .mercurio
```

Esto nos permite limitar nuestros planetas a los 8 que hemos definido en el enum. De esta forma, podemos trabajar de forma más limpia, rápida y con menos errores.

Si lo prefieres puedes declarar las enums de la siguiente forma. Se trata de una sintaxis más clara que puede ayudar a la comprensión de tu código:

```
enum planet{  
    case mercurio  
    case venus  
    case tierra  
    case marte  
    case jupiter  
    case saturno  
    case urano  
    case neptuno  
}
```

Eres libre de elegir la sintaxis con la que te encuentres más cómodo.

Como puedes ver, los enums, pueden sernos muy útiles cuando tenemos un grupo de valores que queremos mantener agrupados y únicamente utilizaremos un valor del grupo cada vez.

Hay que decir, que las enumeraciones tienen dos tipos diferentes de valores: La descripción y el valor interno o **hashValue**.

Nosotros normalmente trabajaremos siempre con la descripción, pero internamente el sistema les asigna un hashValue y así es como reconoce cada elemento.

El hashValue es el índice interno que se asigna de forma automática a cada uno de los elementos que forman la enum.

Siguiendo con el ejemplo anterior:

```
planet.mercurio.hashValue  
planet.tierra.hashValue  
planet.neptuno.hashValue
```

0
2
7

Como puedes ver, al tratarse Mercurio del primer elemento del enum le corresponde el hashValue 0. La Tierra es el tercer elemento por lo que le corresponde el 2. Neptuno es el último elemento por lo que le corresponde el 7.

Como puedes ver, estos valores los asigna directamente el enum, basándose en el orden en el que hemos añadido los elementos.

Existe la posibilidad de que seamos nosotros mismos quienes especifiquemos un valor interno que queremos que tenga cada objeto. Además este valor interno no tiene porque ser numérico, puede ser cualquier de estos tipos:

- Int
- Float
- String
- Character

Imagina que en lugar de que su índice interno comience en 0, como ahora sucede, queremos que comience en 1. Tendríamos que hacer lo siguiente al declarar el enum:

```
enum planet: Int {  
    case mercurio = 1, venus = 2, tierra = 3, marte = 4, jupiter = 5, saturno = 6, urano = 7,  
    neptuno = 8  
}
```

Para poder asignarle el valor interno a cada uno de los elementos, tendremos que especificar el tipo del valor interno que queremos asignarle. En nuestro caso, como queremos asignarle valores del 1 al 8, hemos especificado en nuestra enum, el tipo Int.

Si en lugar de valores numéricos quisiéramos asignarle cadenas de texto a cada elemento, al declarar la enum tendríamos que haber especificado tipo String.

Ahora, para obtener cada uno de estos valores internos, en lugar de utilizar la propiedad `hashValue`, utilizaremos `rawValue`, que es la que hace referencia al valor interno creado por nosotros:

```
planet.mercurio.hashValue  
planet.mercurio.rawValue
```

0
1

Puedes ver la diferencia entre el valor interno generado de forma automática (`hashValue`) que muestra un 0 y el valor interno generado por nosotros mismos (`rawValue`) que muestra un 1.

De hecho, como le hemos asignado valores numéricos consecutivos a nuestra enum, podríamos haberla declarado omitiendo los valores posteriores al 1 y el compilador será capaz de llenar de forma secuencial el resto de valores:

```
enum planet: Int{  
    case mercurio = 1, venus, tierra, marte, jupiter, saturno, urano, neptuno  
}
```

Podemos utilizar nuestras enums junto con un switch para poder realizar diferentes operaciones en función del valor de nuestro enum.

Por ejemplo, imagina que en nuestra aplicación del Sistema Solar queremos mostrar una pequeña descripción de cada uno de los planetas, en función del planeta que elija el usuario.

Para ello podríamos utilizar nuestra enum, junto con un switch para mostrar por pantalla el mensaje correspondiente a cada planeta.

```
var planetSelected = planet.tierra

switch planetSelected{

case .mercurio: print("Es el planeta más pequeño de todos")
case .venus: print("Venus no tiene satélites naturales")
case .tierra: print("El único planeta en el que podemos vivir. Por ahora...")
case .marte: print("Primer planeta que colonizará el ser humano")
case .jupiter: print("El planeta más grande de todos")
case .saturno: print("El único con un sistema de anillos visible desde la Tierra")
case .urano: print("Es el primer planeta descubierto mediante un telescopio")
case .neptuno: print("Se trata del planeta con los vientos más fuertes")

}
```

OTROS USOS DE LOS ENUM:

Por ejemplo, podrían utilizarse para almacenar las diferentes URLs que tengamos de nuestro backend. Ya que se trata de un grupo de elementos comunes (Diferentes URLs) y lo normal es que únicamente accedamos a uno diferente cada vez.

Además los enum tienen sus propios métodos que los convierten en algo muy potente.

EXTENSIONES

WWW.EFECTOAPPLE.COM

El siguiente concepto es junto con los Structs y los Protocolos, uno de los más usados en Swift.

Estamos hablando de las **Extensiones**.

Si tienes experiencia con Objective-C, recordarás el concepto de Categorías.

Pues las extensiones en Swift son muy similares a las categorías de Objective-C.

Y, ¿para qué sirven las extensiones?

Las extensiones nos permiten añadir funcionalidades nuevas a clases preexistentes. Pero no solo a clases, también a Enums y a Structs.

De forma simple podemos decir que nos permiten añadir cosas pero no cambiar las que ya existen.

La sintaxis de una extensión sería la siguiente:

```
extension TipoAExtender{
    //Declaramos todo lo que queramos añadir al Tipo Preexistente
}
```

Además puedes aplicar un protocolo a esa extensión:

```
extensión TipoAExtender: Protocolo{
    //Aquí iría nuestro protocolo implementado
}
```

Si te das cuenta, no es necesario que nuestra extensión tenga un nombre. Simplemente haremos referencia al elemento (Clase, Struct o Enum) que queramos extender y a continuación declaramos todo lo que queramos añadirle.

A continuación vamos a ver un ejemplo de como añadir funciones a una clase preexistente utilizando extensiones.

Imagina que estás trabajando en un proyecto, en el que tienes que realizar mediciones de temperatura. En este proyecto tienes que realizar conversiones entre grados Celsius y grados Fahrenheit. Has creado dos funciones para realizar estas conversiones: Una que pase de Celsius a Fahrenheit y otra que pase de Fahrenheit a Celsius:

```
func celsiusToFahrenheit() -> Double {  
    return self * 9 / 5 + 32  
}  
  
func fahrenheitToCelsius() -> Double {  
    return (self - 32) * 5 / 9  
}
```

Estas dos funciones las tendrás que utilizar mucho a lo largo de tu proyecto, por lo que la opción recomendada será utilizar una extensión para añadirlas por ejemplo a la clase Double. De esta forma, cuando declares cualquier variable de tipo Double, podrás acceder fácilmente a estos métodos para realizar las conversiones.

Veamos como creariamos una extensión de Double, que añadiera estos dos métodos:

```
extension Double {  
  
    func celsiusToFahrenheit() -> Double {  
        return self * 9 / 5 + 32  
    }  
  
    func fahrenheitToCelsius() -> Double {  
        return (self - 32) * 5 / 9  
    }  
}
```

Como puedes ver, de forma muy simple hemos añadido dos funciones de utilidad para nuestro proyecto a la clase preexistente Double.

Cada vez que queramos realizar una conversión, por ejemplo, de Celsius a Fahrenheit podremos utilizar nuestra extensión:

```
//Declaramos una var de tipo Double y le asignamos el valor 100.0  
let boilingPointCelsius = 100.0  
//Pasamos de Celsius a Fahrenheit utilizando el método de nuestra extensión  
let boilingPointFahrenheit = boilingPointCelsius.celsiusToFahrenheit()
```

100

212

De forma muy simple, acabamos de realizar la conversión.

Además de funciones, también podemos añadir propiedades computadas a clases preexistentes.

Imagina que queremos añadir una variable computada a la clase Int que realice el cálculo de elevar al cuadrado un número cualquiera. Por ejemplo, podríamos llamar a la variable computada: squared. Lo haríamos de la siguiente forma:

```
extension Int {  
    var squared: Int {  
        return(self * self)  
    }  
}
```

Así, cada vez que utilicemos la propiedad computada squared, nuestro número será elevado al cuadrado:

6. squared

36

Como has podido comprobar, las extensiones, son una característica muy útil de Swift que nos permite aumentar las funcionalidades del lenguaje y facilita nuestra labor como desarrolladores.

PROTOCOLOS

WWW.EFECTOAPPLE.COM

Swift es un lenguaje que pone especial énfasis en los Protocolos. De hecho, se podría decir que en lugar de un lenguaje orientado a objetos se trata de un lenguaje orientado a protocolos.

Sin ir más lejos, la librería estándar de Swift está construida con protocolos.

En lugar de trabajar con clases y superclases, Swift está más enfocado en el uso de protocolos para ampliar las funcionalidades de nuestras clases. Se podría decir, que son la opción por defecto para extender el lenguaje.

La diferencia entre un paradigma y otro es que en un lenguaje orientado a objetos, trabajamos modificando nuestras clases y creando nuevas clases derivadas, mientras que en un lenguaje orientado a protocolos, nuestras clases permanecen inalteradas y gracias a los protocolos que vamos definiendo, somos capaces de añadirles nuevas funcionalidades.

Además de los protocolos que nosotros vayamos creando, siempre podremos aprovechar una gran cantidad de protocolos que Swift nos ofrece para su uso.

La sintaxis de un protocolo en Swift sería la siguiente:

```
protocol ExampleProtocol: OtherExampleProtocol{  
    var readWriteProperty: String {get set}  
    var readOnlyProperty: String {get}  
  
    func function_name() -> Int  
}
```

Como puedes ver en la sintaxis, se utiliza la palabra reservada *protocol*, seguida del nombre del protocolo y después del nombre se pueden añadir otros protocolos, que implemente nuestro protocolo actual.

De esta forma, la clase que implemente *ExampleProtocol*, implementará también *OtherExampleProtocol*.

Después, entre llaves aparecen las propiedades y funciones que queramos definir en nuestro protocolo. En este caso, hemos declarado dos variables y una función.

Fíjate que hemos definido la primera propiedad de lectura y escritura y la segunda únicamente de lectura.

Para hacer esto simplemente tendrás que añadir entre llaves {get set} en el primer caso y solo {set} si quieres que tu propiedad sea de solo lectura. De esta forma, nadie desde fuera de la clase podrá modificar su valor.

Si quisiéramos declarar una clase que conformara este protocolo haríamos lo siguiente:

```
class ExampleClass: ExampleProtocol {  
}
```

Si recuerdas los protocolos en Objective-C, existía la posibilidad de marcar elementos opcionales y otros obligatorios dentro de un protocolo.

Esto en Swift puro no puede hacerse. Lo que si puedes hacer es añadir la palabra reservada `@objc`, que hará que nuestro protocolo sea compatible con Objective-C y podamos definir los elementos de nuestro protocolo que queramos que sean opcionales con la palabra reservada `optional`.

**Recuerda que deberás hacer
import Foundation en tu playground para poder utilizar Objective-C**

Aquí tienes un ejemplo:

```
import Foundation  
  
@objc protocol ExampleProtocol {  
  
    optional var readWriteProperty: String {get set}  
    var readOnlyProperty: String {get}  
  
    optional func function_name() -> Int  
}
```

En este caso habríamos declarado como opcionales la propiedad `readWriteProperty` y la función `function_name`

Yo, como en apartados anteriores te recomiendo que utilices Swift puro siempre que te sea posible y no andes añadiendo compatibilidades con Objective-C en tu código.

GENÉRICOS

WWW.EFECTOAPPLE.COM

Swift es un lenguaje de programación moderno que nos ofrece muchas ventajas frente a nuestro viejo amigo Objective-C.

Sin embargo, como cualquier otro lenguaje, Swift no es perfecto.

Como has podido ir viendo a lo largo de los conceptos que hemos ido explicando, es un lenguaje bastante estricto en lo relacionado a trabajar con diferentes tipos de datos al mismo tiempo.

Sin ir más lejos, en una colección de Swift, únicamente puedes almacenar objetos del mismo tipo.

Otro ejemplo podría ser cuando trabajamos con funciones y especificamos los tipos de sus parámetros y el tipo de dato que devuelven. Si queremos utilizar otros tipos de datos con esta función no podremos hacerlo.

Esto genera una serie de problemas, que vamos a ver con un ejemplo muy claro.

Imagina que desarrollamos una función muy simple. Queremos que reciba como parámetro 2 objetos Int y devuelva una tupla (Int, Int) con esos dos objetos:

```
func makeIntPair (x: Int, y: Int) -> (Int, Int){  
    return (x,y)  
}
```

Si queremos llamar a nuestra función, lo haremos pasándole los 2 parámetros de tipo Int:

```
makeIntPair(10, y: 20)
```

(.0 10, .1 20)

Pero...

¿Qué ocurre si queremos utilizar esta función pasándole dos parámetros de tipo String en lugar de los parámetros de tipo Int?

Lo que ocurre es que recibimos un error de compilación.

```
makeIntPair("Homer", y: "Marge") ❌ Cannot convert value of type 'String' to expected argument type 'Int'
```

Con los conceptos que hemos visto hasta ahora, Swift no nos permite crear una función “genérica” que acepte diferentes tipos de datos.

Esto nos obligaría a crear una función específica para trabajar con Strings:

```
//Implementamos nuestra función que trabaja con Strings
func makeStringPair (x: String, y: String) -> (String, String){
    return (x,y)
}
//Llamamos a nuestra función
makeStringPair("Homer", y: "Marge")
```

Si para cada tipo de dato, tenemos que generar una función diferente con el mismo código, lo que vamos a obtener es una cantidad enorme de código repetido y además muchas veces vamos a trabajar por duplicado y triplicado.

Esto va totalmente en contra de los principios fundamentales de la programación.

Así que para solucionar este problema, Swift nos ofrece **Los Genéricos**.

Los genéricos nos permiten declarar funciones de forma general, es decir, funciones que pueden aceptar diferentes tipos de datos.

Utilicemos el ejemplo anterior para demostrar como funcionan los genéricos.

Si quisiéramos crear una función genérica que reciba 2 parámetros y devuelva una tupla con los 2 parámetros, sin importar el tipo de los parámetros, haríamos esto:

```
func makePair<T> (x: T, y: T) -> (T, T){
    return (x, y)
}
```

Utilizamos la letra **T** para especificar que los tipos de los parámetros son genéricos.

Si quisiéramos especificar más parámetros genéricos, utilizaríamos: **U, V, W, X, Y, Z**. Esto lo veremos con otro ejemplo, más adelante.

Como ves, se marcan entre <> los tipos de los parámetros que vayamos a utilizar y luego se asignan a cada uno de los parámetros. T también se utiliza en el tipo de retorno.

Después de haber declarado esta función, ya podremos utilizarla para trabajar con objetos de tipo Int:

```
makePair(10, y: 20)
```

(.0 10, .1 20)

Y con objetos de tipo String:

```
makePair("Homer", y: "Simpson")
```

(.0 "Homer", .1 "Simpson")

Realmente, lo que hace el compilador cuando llamamos a una función genérica es crear una versión específica del código, especializada para ese tipo concreto. Por lo que trabaja con diferentes funciones, mientras nosotros solo tenemos que crear una única función. Esto es perfecto para el desarrollador, ya que nos permite delegar este trabajo en el compilador.

¿Qué ocurre si intentamos utilizarla combinando Int con String?

```
makePair("Homer", y: 20)
```

! Cannot invoke 'makePair' with an argument list of type '(String, y: Int)'

Obtendremos un error de compilación.

Esto se debe, a que al declarar nuestra función genérica, hemos especificado con **T**, que ambos parámetros, aunque son genéricos, son del mismo tipo y como le estamos pasando un String y un Int, esto no es cierto.

Para trabajar con objetos de tipo Int y String al mismo tiempo tendríamos que haber declarado la función de esta otra forma:

```
func makePair<T,U> (x: T, y: U) -> (T, U){  
    return (x, y)  
}
```

Así, estamos especificando que ambos parámetros pueden ser de tipos genéricos y diferentes entre ellos, a través de las letras **T** y **U**.

Al declararla de esta forma, ya podremos realizar la llamada pasándole un String y un Int:

```
makePair("Homer", y: 20) (.0 "Homer", .1 20)
```

Acabamos de ver como podemos declarar funciones que acepten tipos genéricos.

Pero no solo las funciones pueden trabajar con genéricos. También las Clases, las Structs y las Enums pueden ser genéricos.

También podemos utilizar genéricos con las colecciones de Swift.

Por ejemplo, si declaramos un array de tipo Int y le asignamos un Float, obtendremos un error de compilación:

```
var elements: Array<Int> = [100, 200, 500, 1000, 5000.50]  
⚠ Cannot convert value of type 'Double' to expected element type 'Int'
```

Esto en principio nos limitaría bastante, si no fuera porque gracias a los genéricos podemos crear tipos genéricos que nos permitirán almacenar diferentes tipos de datos.

¿Cómo podemos hacerlo?

Haciendo que la colección sea de un tipo de dato que conforme un protocolo concreto.

Así podríamos guardar en una colección diferentes tipos que implementen el mismo protocolo.

Por ejemplo, si creo un array de elementos que conformen el protocolo **<CustomStringConvertible>**, le podré añadir tanto objetos de tipo Int, como otros arrays:

```
var newElements: Array<CustomStringConvertible> = [100, [100, 200, 300]]
```

DESPEDIDA

WWW.EFECTOAPPLE.COM

Hemos llegado al final del libro en el que hemos visto concepto a concepto una introducción al lenguaje Swift.

Espero que todo lo que hemos visto en este ebook te haya sido muy útil. He puesto todo de mi parte para que así sea.

Me despido. Tan solo me queda decirte GRACIAS, por tu confianza, por tu tiempo y por el interés en aprender Swift.

Espero de verdad que este libro haya cumplido tus expectativas de formación y conocimiento, pero si no fuera así, desde aquí te brindo mi ayuda en lo que necesites, no dudes en escribirme.

Puedes contactar conmigo por email en behappy@efectoapple.com.

También te quiero recordar que puedes seguir en contacto conmigo a través de mi blog www.efectoapple.com, twitter y facebook.

Estas últimas líneas las quiero emplear en decirte que, si te ha gustado el libro y piensas que es útil, te estaría MUY AGRADECIDO si dejas tu opinión en el "Review" (Solo si has comprado este ebook desde Amazon)

Tu apoyo es muy importante para mí.

Leo todos los comentarios e intento mejorar este libro y futuras formaciones.

Te mando un fuerte abrazo.

Luis.-