Aprendiendo Swift

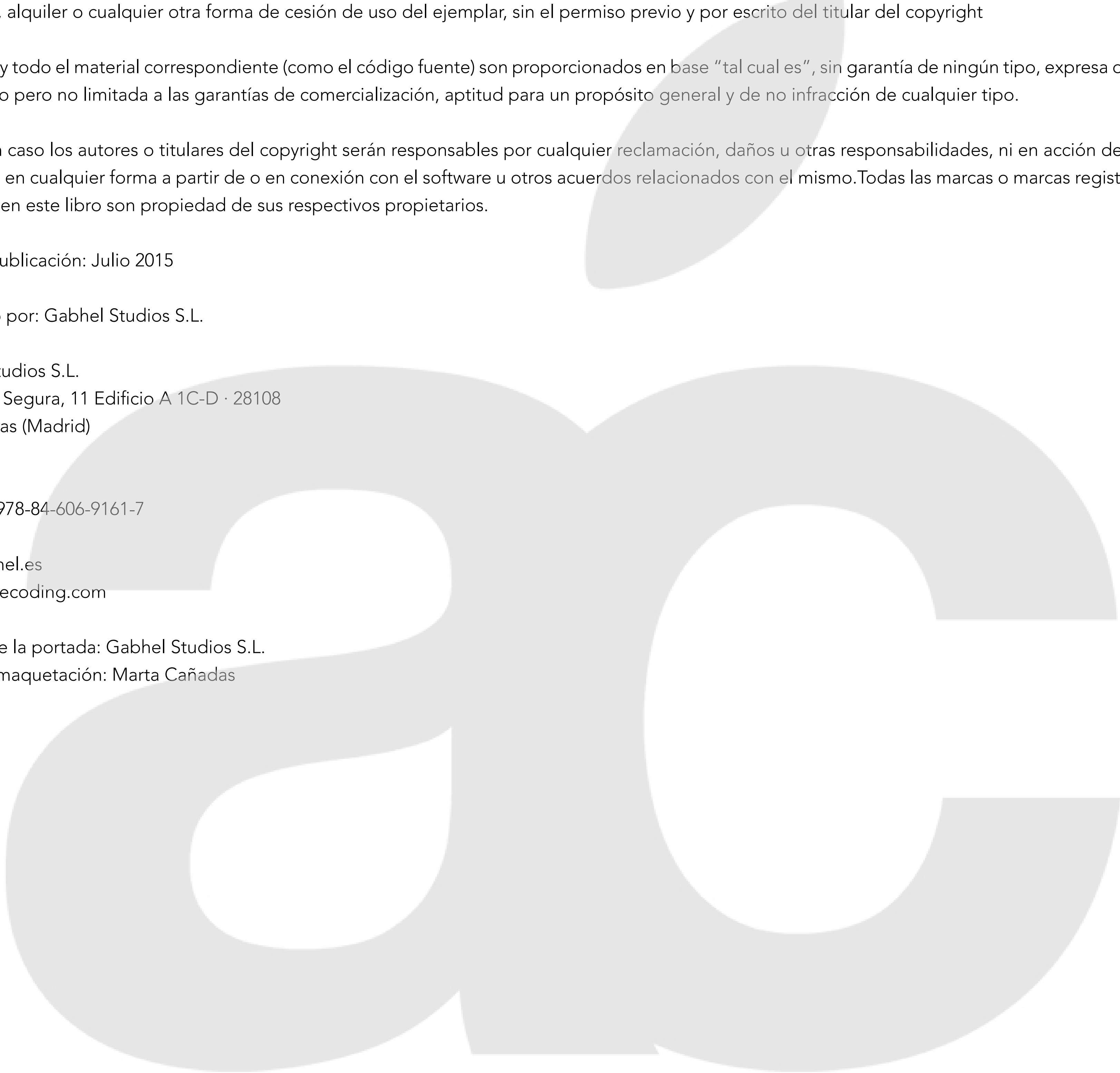
El nuevo lenguaje para iOS y OS X

Copyright © 2015 Gabhel Studios S.L. Todos los derechos reservados

Todos los derechos reservados. Sin limitar el alcance de la reserva de derechos, se informa de que no está permitida la reproducción, distribución, comunicación pública (incluida la puesta a disposición del público) ni transformación en ninguna forma o por cualquier medio, ya sea electrónico, mecánico o por fotocopia, por registro u otros métodos, de todo o parte de este libro electrónico o sus materiales correspondientes (como textos, imágenes o código fuente), ni su préstamo, alquiler o cualquier otra forma de cesión de uso del ejemplar, sin el permiso previo y por escrito del titular del copyright

Este libro y todo el material correspondiente (como el código fuente) son proporcionados en base "tal cual es", sin garantía de ningún tipo, expresa o implícita, incluyendo pero no limitada a las garantías de comercialización, aptitud para un propósito general y de no infracción de cualquier tipo.

En ningún caso los autores o titulares del copyright serán responsables por cualquier reclamación, daños u otras responsabilidades, ni en acción de contrato, agravio, o en cualquier forma a partir de o en conexión con el software u otros acuerdos relacionados con el mismo.Todas las marcas o marcas registradas que aparecen en este libro son propiedad de sus respectivos propietarios.



Primera publicación: Julio 2015

Publicado por: Gabhel Studios S.L.

Gabhel Studios S.L.
C/Anabel Segura, 11 Edificio A 1C-D · 28108
Alcobendas (Madrid)
España

ISBN-13: 978-84-606-9161-7

www.gabhel.es
www.applecoding.com

Imagen de la portada: Gabhel Studios S.L.
Diseño y maquetación: Marta Cañadas

Aprendiendo Swift

El nuevo lenguaje para iOS y OS X

Contenido

1 - INTRODUCCIÓN

1.1 Estructura del libro

1.2 Breve historia del desarrollo en entornos Apple

1.3 Swift, un nuevo lenguaje de programación creado por Apple

1.3.1 ¿Por qué Swift? Un breve repaso histórico

1.3.2 ¿Qué es Swift?

1.3.3 Pilares básicos de Swift

1.3.4 Conclusiones

2 - INTRODUCCIÓN A LA PROGRAMACIÓN

2.1 Algoritmo básico: entrada, proceso y salida

2.2 Algoritmos de control de flujo

2.3 Elementos básicos de la programación orientada a objetos

2.3.1 Concepto de programación orientada a objetos

2.3.2 Herencia de objetos

2.3.3 Redefinición

2.3.4 Polimorfismo

3 - PLAYGROUNDS, PROTOTIPOS EN SWIFT

3.1 ¿Qué es un playground?

3.2 Probando con algo de código

3.3 Prototipos de apps y juegos

3.3.1 Prototipo de una app

3.3.2 Prototipo de un juego

3.4 Usando y creando documentación

3.4.1 Usando la documentación de Xcode

3.4.2 Creando documentación

3.5 Swift R.E.P.L.

4 - SWIFT BÁSICO

4.1 Tipos de datos

4.1.1 Variables y constantes

4.1.2 Tipos por valor o por referencia

4.1.3 Tipos de datos no vacíos

4.1.4 Tipos de datos básicos

4.1.5 Inferencia de tipos

4.1.6 Alias de tipos

4.2 Opcionales

4.3 Operadores (I): Asignación, aritméticos, comparación y rangos

4.3.1 Operador de asignación

4.3.2 Operadores aritméticos

4.3.3 Operadores comparativos

4.3.4 Operadores de rango

4.4 Controles de flujo

4.4.1 Condiciones con if y else

4.4.2 Switch, sentencias case

4.4.3 Bucles for

4.4.4 Bucle while

4.4.5 Control de transferencia y etiquetas

4.5 Operadores (II): Operadores lógicos

4.5.1 Operador de coalescencia nula

4.6 Cadenas
- 4.6.1 Cadenas y caracteres
- 4.6.2 Interpolación
- 4.6.3 Concatenando
- 4.6.4 Contando y comparando
- 4.6.5 Trabajando con subcadenas
- 4.6.6 Procesando las cadenas con subfunciones
- 4.6.7 Uso de caracteres Unicode

4.7 Colecciones (I): Arrays o Matrices
- 4.7.1 Accediendo e inicializando
- 4.7.2 Añadiendo y quitando elementos
- 4.7.3 Buscando y enumerando
- 4.7.4 *Arrays* de más de una dimensión
- 4.7.5 Una cadena también es un *array*

4.8 Colecciones (II): Diccionarios
- 4.8.1 Creando e inicializando
- 4.8.2 Accediendo a los datos
- 4.8.3 Añadiendo y modificando datos

4.9 Colecciones (III): Sets
- 4.9.1 Inicializando los sets y trabajando con ellos
- 4.9.2 Construyendo conjuntos
- 4.9.3 Comparando conjuntos

4.10 Tuplas
- 4.10.1 Descomponiendo una tupla
- 4.10.2 Arrays de tuplas
- 4.10.3 Asignando variables a través de tuplas

4.11 Enumeraciones
- 4.11.1 Inicializando una enumeración
- 4.11.2 Tipificar una enumeración
- 4.11.3 Enumeraciones por valores asociados

4.12 Funciones
- 4.12.1 Usando parámetros
- 4.12.2 Polimorfismo
- 4.12.3 Parámetros externos
- 4.12.4 Parámetros por defecto
- 4.12.5 Parámetros de entrada y salida
- 4.12.6 Funciones variádicas

4.13 Casting o conversión de tipos
- 4.13.1 El problema de las colecciones
- 4.13.2 Casting de tipos, un opcional
- 4.13.3 Comprobando el tipo

4.14 Clases y herencias
- 4.14.1 Inicializadores
- 4.14.2 Inicializadores de conveniencia
- 4.14.3 Herencia
- 4.14.4 Sobrescritura de métodos en herencia
- 4.14.5 Inicializadores obligatorios para subclases

4.15 Structs o estructuras
- 4.15.1 Inicializadores
- 4.15.2 Accediendo a los datos
- 4.15.3 Mutando las propiedades
- 4.15.4 Tipos de dato por valor no por referencia

5 - SWIFT AVANZADO

5.1 Propiedades

5.1.1 Concepto

5.1.2 Propiedades calculadas

5.1.3 Observadores de propiedades

5.1.4 Propiedades perezosas (lazy)

5.1.5 Herencia de propiedades

5.2 Closures

5.2.1 Closures como parámetros en funciones

5.2.2 Otros ejemplos prácticos de uso de un *closure*

5.3 Programación Funcional

5.3.1 Optimizando una función

5.3.2 Primer paso: funciones como parámetros

5.3.3 Segundo paso: aplicando *closures*

5.3.4 Paso tres: reduciendo los *closures* gracias a la inferencia

5.3.5 Paso cuatro: funciones de operador

5.4 Funciones Avanzadas

5.4.1 Funciones anidadas

5.4.2 Funciones genéricas para colecciones: map, filter, reduce y sorted.

5.4.3 Funciones parcializadas

5.5 Extensiones

5.5.1 Concepto

5.5.2 Extendiendo funcionalidades

5.6 Protocolos

5.6.1 Concepto

5.6.2 Creando protocolos

5.6.3 Usando protocolos

5.6.4 Usando protocolos como tipos

5.6.5 Protocolos como tipos de dato

5.6.6 Los protocolos del sistema

5.6.7 Delegación (Delegates)

5.7 Genéricos

5.7.1 Concepto

5.7.2 Entendiendo cómo funcionan

5.7.3 Genéricos que cumplan protocolos

5.7.4 Clases y estructuras genéricas

5.8 Subscripts

5.8.1 Concepto

5.8.2 Ejemplos básicos

5.9 ARC y gestión de memoria

5.9.1 Conteo de referencias

5.9.2 Ciclo de retención: referencias strong y weak

5.9.3 Variables de tipo *unowned*

5.9.4 La importancia de gestionar bien la memoria

5.10 Encadenamiento de opcionales

5.10.1 Concepto

5.10.2 El problema a resolver

5.10.3 Encadenamiento de opcionales, la solución

5.10.4 Encadenamiento de opcionales para subscripts

5.11 Inicializadores falibles

5.11.1 Concepto

5.11.2 Inicializador falible en clases

5.11.3 Inicializadores falibles en structs
5.11.4 Inicializadores falibles en enumeraciones
5.11.5 Inicializadores falibles como prevención
5.12 Operadores personalizados y sobrecarga
5.12.1 Concepto
5.12.2 Operadores personalizados
5.12.3 Sobrecarga de operadores (I): funciones para el nuevo operador
5.12.4 Sobrecarga de operadores (II): funciones para un operador y tipo existente
5.13 Control de acceso
5.13.1 Control de acceso, para proyectos
5.13.2 Internal
5.13.3 Public
5.13.4 Private
5.13.5 Normas de uso en el control de acceso

6 - SWIFT Y OBJECTIVE-C
6.1 Usar Objective-C y C con Swift
6.1.1 Creamos un proyecto en Xcode
6.1.2 Incorporando las cabeceras de C al proyecto en Swift
6.1.3 Configuración manual, incorporando bases de datos SQLite
6.2 Diferenciando Swift de Objective-C
6.2.1 Tipos por valor y por referencia, inicializaciones
6.2.2 Invocar objetos de Objective-C en Swift
6.2.3 Swift y Objective-C, diferentes pero complementarios

7 - REFERENCIAS
7.1 La fundación de Swift
7.1.1 Opcionales
7.1.2 Rangos
7.1.3 Bloques de estructuras de datos
7.1.4 Colecciones, ejemplo de genéricos
7.1.5 Un lenguaje creado sobre sí mismo
7.2 Documentando y comentando
7.2.1 Comentando
7.2.2 Una implementación a mejorar
7.2.3 Documentación simple
7.2.4 Documentación avanzada
7.2.5 Marcas en el código
7.3 Referencias por instrucciones
7.4 Referencias por ejemplos (Swift Básico)
7.5 Referencias por ejemplos (Swift Avanzado)

1 - INTRODUCCIÓN

1.1 Estructura del libro

Bienvenidos a "Aprendiendo Swift", el primero de una serie de libros editados por la web Apple Coding, sobre el desarrollo en entornos Apple y que permite servir de guía de aprendizaje y referencia para todo tipo de perfiles. Desde el profano en el mundo de la programación y que quiere dedicarse a esto o simplemente aprender, pasando por gente de diferente nivel en otros lenguajes, en Objective-C o incluso gente que ya sepa Swift pero quiera tener un libro de referencia del lenguaje.

En el presente libro se abarcará desde unos pequeños tintes históricos en esta misma introducción, pasando por una introducción a la programación y la orientación a objetos, el propio lenguaje Swift, las herramientas para trabajar con él y realizar prototipos, así como cada uno de los elementos imprescindibles del lenguaje.

El libro se dividirá en varios bloques esenciales que permitan seguirlo de una manera más fácil. Primero, en esta introducción haremos un repaso a tres niveles: este que están leyendo, una breve historia en el desarrollo en entornos Apple y por último sobre la breve historia de Swift y por qué Apple decidió lanzar un nuevo lenguaje cuando ya tenía uno (Objective-C) muy asentado y de larga trayectoria.

Tras esto, nos introducimos en Swift, ¿qué es el lenguaje? Para, antes de nada, aprender a cómo vamos a trabajar con todos los ejemplos y código del libro: con el uso de *playground* un nuevo formato de proyecto básico que nos permitirá ver y evaluar todo lo que hacemos. También aprenderemos el uso del REPL del lenguaje o terminal de comandos.

Tras esto, nos remangamos porque comenzamos con el Swift Básico, lo que todo buen programador ha de conocer al dedillo para trabajar con el nuevo lenguaje: tipos, operadores, cadenas, funciones… repasamos todas y cada una de las partes del lenguaje, cómo se usan y lo más importante, ¿para qué usarlas?

Tras superar este capítulo tendremos todas las herramientas en nuestra mano para ser los más preparados desarrolladores en el nuevo lenguaje de Apple, pero eso no es suficiente porque para ser Maestro necesitamos ir más allá. Y para eso está el apartado Swift Avanzado donde conoceremos en mayor detalle los elementos más complejos del lenguaje y que nos permitirán sacar el máximo provecho del mismo y entender cómo funciona.

El siguiente capítulo es un obligado: Objective-C sigue formando parte del ecosistema de Apple, y muchos elementos de las librerías que usamos cada día en Cocoa o SpriteKit, aun siguen programadas en Objective-C. Por lo tanto, es imprescindible que aprendamos cómo interoperar entre los dos lenguajes. Cómo mezclar ambos lenguajes en un solo proyecto, cómo funciona Cocoa… las claves para entender cómo sacar el mejor provecho de ambos.

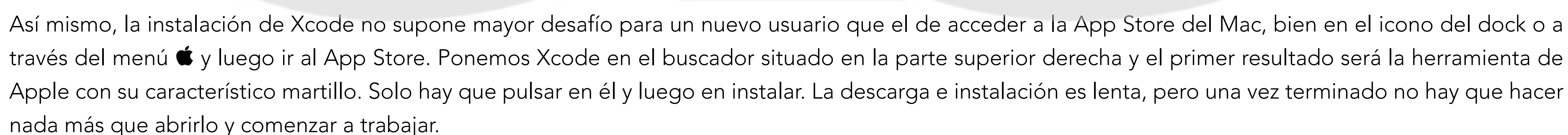
Y por último, las referencias, de consulta imprescindible. Porque es muy complicado tenerlo todo en la cabeza, y esas guías nos ayudan a llegar al grado máximo. Repasamos la fundación del lenguaje para que entendamos su estructura y cómo Swift se construye sobre sí mismo, guías de referencia por ejemplos e instrucciones y cómo documentar nuestro código para trabajar mejor en equipo.

El libro quiere ser una guía de referencia y aprendizaje clara y sencilla, que permita no solo ser un libro de enseñanza si no también de referencia del lenguaje y, por qué no, de la forma de enfrentarnos a los problemas más comunes que se nos pueden presentar con este y cómo solventarlos.

El secreto: explicarlo todo de una manera cercana, evitando los tecnicismos y no dando nada por supuesto. Explicado como te lo explicaría un buen profesor universitario de una manera cercana para que las explicaciones traspasen la complejidad del tema y lleguen en su contenido de una manera especial y cercana, fácil de entender: **hacer fácil lo difícil**.

Para trabajar con Swift hacen falta dos cosas fundamentales: un ordenador Mac con el sistema operativo OS X 10.9 Mavericks o superior y el software para desarrollo Xcode de la propia Apple, en su versión 6.3 o superior, ya que el libro cubre la especificación 1.2 del lenguaje Swift lanzada en abril de 2015.

Si no disponemos de un ordenador Mac, podemos estudiar Swift y aprenderlo, pero lógicamente no podremos poner en práctica lo aprendido. Como IDE (o entorno de desarrollo interactivo) Xcode ofrece todas las herramientas necesarias para trabajar, desde el editor de código hasta constructores de interfaces, de escenas de juegos, de partículas, control de código… aunque podamos usar alguna herramienta externa de ayuda en algún momento, Xcode será lo único que realmente necesitaremos instalar en nuestro Mac.



Así mismo, la instalación de Xcode no supone mayor desafío para un nuevo usuario que el de acceder a la App Store del Mac, bien en el icono del dock o a través del menú y luego ir al App Store. Ponemos Xcode en el buscador situado en la parte superior derecha y el primer resultado será la herramienta de Apple con su característico martillo. Solo hay que pulsar en él y luego en instalar. La descarga e instalación es lenta, pero una vez terminado no hay que hacer nada más que abrirlo y comenzar a trabajar.

Y un detalle, como pequeño consejo personal: una vez abierto os sugiero ir al menú *Xcode* y luego *Preferences*. Pulsamos en *Downloads* y es conveniente que bajemos toda la documentación: tanto la de iOS, como OS X y Xcode. No nos vendrá mal porque además aprenderemos cómo acceder a ella en cualquier momento.

Y empezamos ya con la faena. Antes de nada agradecerte que hayas decidido aprender con este libro que se ha hecho con mucho esfuerzo, durante meses, reflejando la experiencia real en este nuevo lenguaje y en proyectos de apps y juegos. Un libro que supone el trabajo de muchos meses de recopilación, aprendizaje, pruebas, prácticas, proyectos y mucha ilusión.

Gracias y esperamos que cuando llegues al final sientas como Neo al abrir los ojos, mirar a Morfeo y decir aquello de: "Ya sé Kung-Fú".

4 - SWIFT BÁSICO

4.2 Opcionales

Una de las cosas más importantes de Swift, como hemos comentado, es que **no admite una variable o constante sin valor**. En el lenguaje no existe el valor nulo, vacío, *nil* o *null* que puede existir en otros lenguajes. No es un valor válido. No es un valor, de hecho.

¿Qué hacemos entonces si queremos declarar una variable cuyo valor aun desconocemos? Podríamos pensar en usar la facultad de variables o constantes de ser declaradas pero no inicializadas antes de usarse pero tendríamos un problema. Si lo hacemos en el ámbito de una clase, habremos de darle valor antes de inicializar la propia clase. ¿Y si nuestra variable depende de un valor asignado posterior a la inicialización y por lo tanto no podemos asignar o conocer su valor antes de la inicialización de esta? ¿Hemos de darle un valor aleatorio o de relleno?

¿O qué pasa si queremos tener una variable que sí nos interese en un momento determinado que pueda estar vacía? ¿Cómo lo hacemos entonces?

Para ello Swift incorpora un nuevo tipo de dato para las variables que es esencial para nuestro trabajo: las opcionales.

Un valor opcional es un contenedor que puede almacenar solo dos posibles cosas: nada o algo. Mientras una variable o constante en Swift solo puede almacenar algo pero nunca nada, el opcional es una referencia y no un contenedor en sí. Es un contenedor dentro de otro.

Vamos a imaginar que una variable es una caja donde colocamos algo: una cadena, un número, un objeto… dicha caja siempre ha de tener algo, no puede estar vacía. ¿Cómo hacemos para que pueda estar vacía?

Metemos dentro otra caja que sí pueda estarlo. Dicha caja dentro de la caja bien contendrá el mismo tipo de dato que debería contener la caja más grande donde está metida o bien estará vacía. De esta forma, la caja más grande, la de nuestra variable o constante, siempre tendrá algo dentro.

Para la caja grande, la caja pequeña dentro de ella (el opcional) es un `String` o un `Double` pero internamente puede que esté o puede que no. Hasta que no se mira dentro para hacer algo con el dato para el sistema hay algo y no lo hay a la vez como en la paradoja del gato de Schrödinger.

> **Curiosidad:** La paradoja del gato de Schrödinger es un experimento (teórico) de la mecánica cuántica por el que dentro de una caja se pone un gato, un aerosol venenoso y una partícula radiactiva que pasado un tiempo tiene un 50% de probabilidades de desintegrarse y activar el veneno. Pasado este tiempo y aplicando los principios de la mecánica cuántica, si no abrimos la caja, el gato dentro de ella tiene el estado de vivo y muerto a la vez. Por eso es una paradoja ya que según la mecánica cuántica el gato está muerto y vivo al mismo tiempo desde que no podemos discernir su estado salvo rompiendo las condiciones del experimento. Las opcionales para el sistema tienen una curiosa y similar premisa

Las opcionales solo pueden usarse con un tipo variable ya que en su propio concepto son tipos de datos que están destinados a ser creados sin el dato concreto que posteriormente queramos introducirles, lo que va en contra del concepto mismo de un tipo constante. De igual manera no podemos inferir el tipo de dato y hay que indicarlo, en este caso con un signo de interrogación de cierre ? seguido al tipo de dato, como vemos en el ejemplo:

```
var variable: String?                                    nil
```

El por qué no se puede inferir el tipo de dato es claro: el valor vacío `nil` puede ser aplicado a cualquier tipo de dato por lo que si pudiéramos usar inferencia al sistema le resultaría imposible determinar a qué tipo de dato correspondería este valor.

Si miramos la barra de evaluación del *playground* vemos algo que nos llama la atención: el valor de `variable` es `nil`. Esto significa que el sistema la ha inicializado con el valor vacío. Si hacemos lo mismo con un número también pondrá `nil` porque significa que hay ausencia de valor no un valor como 0, por ejemplo, que ya es de por sí un valor. Indica que no hay número.

Cuando añadimos el símbolo de interrogación ? tras el tipo de dato u objeto lo que estamos indicando es una inicialización implícita de este tipo que correspondería a hacer esto:

```
var variable: String? = nil
```

Indicando la interrogación es como podemos asignar `nil` como valor válido de una variable. Si quitamos este, veremos que da un error que en este momento resultará algo críptico, indicando que el tipo no se ajusta a un protocolo. Por lo tanto, dejamos el interrogante y podemos quitar el `= nil`.

```
var variable: String?
```

Vamos ahora a darle un valor en una nueva línea a nuestra variable opcional, haciéndolo como si fuera una variable normal y vemos qué pasa:

```
variable = "Hola opcional"
```

Miramos la evaluación de dicha expresión y nos encontraremos con algo curioso.

```
variable = "Hola opcional"                                    {Some "Hola opcional"}
```

Nos dice que su valor es `{Some "Hola opcional"}`. Esto significa que su valor es literalmente "algo" y luego el valor que hemos dado. Esto es lo que indica que estamos mirando en la caja dentro de la caja. Miramos en la caja, hay algo (lo indica) y luego dice el algo que hay.

```
variable
```

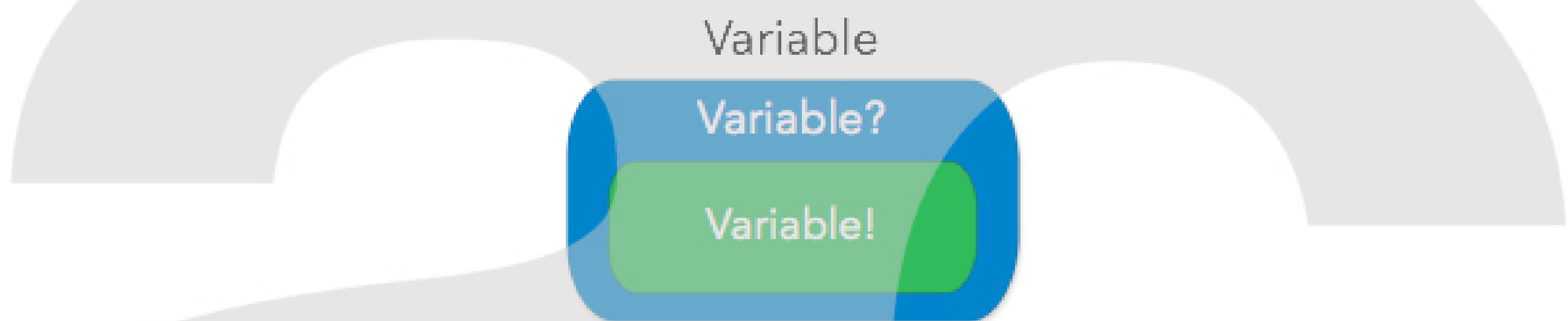
Si ponemos debajo simplemente el nombre para que evalúe la expresión, `variable`, veremos que repite la misma expresión. Esto es debido a que variable, al ser de tipo `String?` usa de manera implícita el ? para referirse a esta. Si ponemos como expresión a evaluar solo `String?` veremos claramente que es igual ponerlo con o sin ?. Ambas expresiones significan lo mismo.

¿Cómo accedemos entonces al valor tal cual que hay en la caja de la opcional? Para hacer como si se tratara de una caja cerrada tenemos que desempaquetarla para ver qué hay dentro. Para ello usamos el símbolo `!` que hace un "desempaquetado" (en inglés *unwrap*) del valor de la caja del opcional y nos muestra lo que hay dentro de ella.

```
variable!
```

Vemos que en la barra de evaluación de expresiones nos da el valor `"Hola opcional"` y ya no nos indica el `Some` de antes ni las llaves de apertura o cierre. Esa es una de las formas de acceder al valor de una variable opcional: desempaquetarla de manera implícita poniendo el símbolo `!` que permite mirar dentro y sacar lo que hay.

Si buscamos un símil aun más claro, `variable` hace referencia a la caja principal, donde no puede existir en ningún momento un valor vacío. Añadiendo el símbolo `?` lo que tendríamos es `variable?` o la referencia a la caja que metemos dentro de `variable` donde puede o no haber algo. Lo que haya dentro de `variable?` (la caja dentro de la caja) es lo que llamaríamos `variable!`.



Pero ojo, hacer un *explicit unwrap* o desempaquetado implícito de un opcional tal cual nos puede traer consecuencias desagradables si no estamos seguros que lo que haya en el opcional es algo y no nada, ya que el sistema no puede saber si al desempaquetar el opcional va a ver algo o no de manera directa.

Es decir, podemos tener un error fatal del programa porque el propio *unwrap* supone coger el contenido (o no contenido) de `variable!` y ponerlo en una variable convencional para ser evaluado. Aunque nosotros solo veamos el nombre, en realidad está desempaquetando el contenido y poniéndolo en una variable del mismo tipo, no opcional. Y como el lenguaje no admite el valor vacío como válido en una variable no opcional, provocará un error fatal del sistema y la app dejará de funcionar inmediatamente.

Vamos a probar a borrar la línea `variable = "Hola opcional"` y veamos qué pasa.

```
var variable: String?                                              nil
variable!                                                          error
        Execution was interrupted, reason: EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0).
```

Eso en nuestro programa significa un cierre no controlado de la aplicación. Se interrumpe la ejecución porque hemos accedido al valor de un opcional que estaba vacío, es decir, hemos intentado meter un valor vacío dentro de una caja normal que ya sabemos que no puede contener valores vacíos. Por eso el cierre.

Cómo era esto en Objective-C

En Objective-C existía la posibilidad que un objeto fuera nil, es decir, que fuera nulo. Y podíamos preguntar si una variable de tipo objeto apuntaba a algo vacío o no. El *nil* en Objective-C es la ausencia de objeto dentro de un tipo objeto, indicando un puntero a un objeto que no existe. Pero este concepto solo era aplicable a los objetos y no a cualquier tipo. En Swift, este concepto se extiende más allá permitiendo que cualquier tipo, sea el que sea, pueda tener el estado de "ausencia de valor" o *nil*. En Swift, un opcional como *nil* especifica implícitamente la ausencia de valor y como tal, *nil* no puede ser aplicado a un objeto sino solo a un opcional de dicho objeto.

¿Cómo controlar entonces si no tenemos claro si un opcional tendrá o no valor y evitar el error? ¿Cómo saber si nuestro gato está vivo o muerto sin abrir la caja? A través de lo que se denominan enlaces opcionales u *optional bindings*.

Como hemos visto, un opcional inicializado responde al evaluarse con `nil`, lo que indica que dentro tiene una caja con ausencia de valor. Mientras intentemos meter el valor de este opcional en una variable que no lo sea, no habrá ningún problema pero si lo queremos introducir es cuando tenemos el problema. La ventaja del `nil` es que sí responde a una condición booleana, es decir, por defecto si preguntamos por un opcional y este es `nil` nos devolverá el valor ***false*** y si tiene un dato que no sea vacío devolverá ***true***.

Aprovechando por lo tanto esta facultad, el procedimiento del enlace opcional nos permite asignar a una constante normal (no una variable) un valor opcional dentro de una condición. Y si esta asignación ha podido realizarse correctamente porque lo que tiene el opcional no es `nil`, nos devolverá un valor true y nos validará la condición haciendo además la declaración y asignación de dicha variable ya desempaquetada como valor normal.

```
var opcional: String?

if let variable = opcional {
    println("Tiene el valor \(variable)")
} else {
    println ("La opcional no contiene valor")
}
```

Como vemos en el ejemplo, definimos una variable pero no le damos valor. ¿Qué pasará al hacer la asignación de valor en la condición `if`? Como la constante `variable` no puede contener un valor que es `nil` no entraría en la condición verdadera y pasaría por el `else` que informa que la opcional no contiene valor. Lo más importante es que tenemos una declaración en línea y que nuestro código no sufre error alguno.

```
var opcional: String?                          nil

opcional?                                      nil

if let variable = opcional {
    println("Tiene el valor \(variable)")
} else {
    println ("La opcional es nil")             "La opcional es nil"
}
```

Sin embargo, cuando sí le damos un valor a `opcional`, vemos que el código se comporta de una manera totalmente diferente.

```
opcional = "No estoy vacío"                    {Some "No estoy vacío"}

if let variable = opcional {
    println("Tiene el valor \(variable)")      "Tiene el valor No estoy vacío"
} else {
    println ("La opcional es nil")
}
```

Al darle valor a `opcional` su valor evaluado es `{Some "No estoy vacío"}`, lo que indica que estamos metiendo el valor de dicha cadena en el opcional el cual detecta que hay algo: de ahí el *some*. Pero como hemos dicho antes, **el enlace opcional desempaqueta el mismo para guardarlo en la constante**. De esta manera nos evitamos usar la `!` si no que directamente, si la condición es correcta porque `opcional` tiene un valor, lo que irá en `variable` será el valor tal cual. Usar el `if` desempaqueta automáticamente y mete el valor en la variable.

Esta práctica es la más recomendada a la hora de trabajar con opcionales y extraer su valor, a no ser que tengamos perfectamente claro que bajo ningún concepto, desempaquetar un opcional implícitamente con `!` nos podría dar un valor vacío que provoque un error fatal en nuestro código.

Como nota final, es importante que tengamos en cuenta que `variable` solo podrá ser usada en el ámbito del `if` en caso que este sea verdadero. Fuera de ese ámbito, `variable` no existirá. En un próximo capítulo sobre controles de flujo veremos más en detenimiento el ámbito de actuación de las variables o constantes.

En conclusión: recuperar valores de un opcional a través de una asignación de valor en un `if`, es lo se conoce como *optional binding* o enlace opcional.

Una de las cosas más interesantes que podemos hacer para trabajar con los enlaces opcionales, es la posibilidad de encadenar varios en un mismo `if` con el objeto de optimizar la cantidad de código a utilizar. Si tuviéramos que hacer los enlaces de un conjunto de 4 opcionales, donde además quisiéramos evaluar el valor de una de ellas como una condición más, el código resultante sería el siguiente:

```
var ab:String?
var ac:String?
var ad:Int?
var ae:String?

if let vab = ab {
    if let vac = ac {
        if let vad = ad {
            if vad != 0 {
                if let vae = ae {
                    println ("\(vab) \(vac) \(vad) \(vae)")
                }
            }
        }
    }
}
```

Esta consecución de anidamientos es lo que se conoce comúnmente como "la pirámide del mal", una de las prácticas peor vistas en programación y creación de algoritmos eficientes. En Swift, podemos optimizar este código encadenando cada enlace de opcional separándolo por comas de la siguiente forma:

```
if let ab = ab, let ac = ac, let ad = ad where ad != 0, let ae = ae {
    println ("\(ab) \(ac) \(ad) \(ae)")
}
```

En los enlaces opcionales anidados podemos usar una cláusula `where` que nos permita en cualquiera de dichos enlaces la evaluación de un valor en caso que el enlace de la opcional haya devuelto un resultado válido. De esta forma, podemos integrar condiciones más complejas.

Por estructura de la instrucción, hemos usado el `let` en cada uno de los enlaces por haber puesto el `where` en una condición intermedia. En caso de no usar ninguna cláusula `where`, la instrucción quedaría mucho más sencilla:

```
if let ab = ab, ac = ac, ad = ad, ae = ae {
    println ("\(ab) \(ac) \(ad) \(ae)")
}
```

En el caso que queramos, por ejemplo, evaluar solo una expresión las buenas prácticas nos recomiendan llevar dicho enlace evaluado al último de los anidamientos, quedando la instrucción de la siguiente forma:

```
if let ab = ab, ac = ac, ae = ae, ad = ad where ad != 0 {
    println ("\(ab) \(ac) \(ad) \(ae)")
}
```

La consecución de enlaces opcionales actúa como un operador lógico AND o de verdadero exclusivo, evaluando las expresiones una a una. Solo en el caso que todos los enlaces realizados resulten correctos y devuelvan un valor que no sea vacío se cumplirá la condición y se ejecutará lo que haya dentro de la condición `if`.

En el momento que uno de los enlaces no se cumpla y el opcional tenga un valor vacío la condición dará un resultado negativo y no evaluará el resto de condiciones en caso que las hubiera. De esta forma, si la opcional `ab` no tuviera valor, dejaría de comprobar y no validaría `ac`, `ad` ni `ae`.

¿Por qué opcionales?

¿Se preguntará más de uno? ¿No es mucho lío esto de las opcionales? ¿No sería más fácil permitir valores vacíos? ¿No me sirve en todos los casos el poder declarar la variable o constante aunque no le de valor al principio y darle dicho valor después?

Bien, podría ser más fácil, desde luego, pero no más seguro y no aplicable a todos los casos. El tipo opcional tiene una función muy clara: **asegurar una mejor calidad del código por parte de los programadores y evitar errores mayores.**

En otros lenguajes, como Objective-C, una variable vacía no es un error. Esto significa que si nuestro programa tiene un error en su ejecución porque una variable o un objeto se queda vacío este será difícil de detectar ya que dicho error podría generar comportamientos no esperados pero que no se consideren erróneos como parte del programa en sí porque no se consideran negativos para su funcionamiento.

Es el concepto del *glitch* que es algo más difícil de localizar y corregir que un *bug*. Mientras un *bug* es un error que afecta negativamente a nuestro programa, un *glitch* es un comportamiento no deseado que sin embargo no afecta al rendimiento o estabilidad.

Con el objetivo de conseguir un código de mejor calidad y más seguro, que prevenga de estos problemas, Swift usa las opcionales con el objeto que aquellas variables que puedan ser vacías tengan una forma diferente de ser tratadas trasladando al desarrollador la responsabilidad de tener un mayor control de las mismas.

Lo que hace es convertir en error grave de ejecución que una variable no tenga valor, lo que a la larga traslada errores comunes a momentos de desarrollo y no de ejecución, como suele ser habitual en muchos casos. Los errores graves durante el desarrollo afectan y obligan al desarrollador a hacer mejor su trabajo mientras que los errores graves en momentos de ejecución afectan al usuario final y dan una mala imagen de la calidad de las apps.

Además, no en todos los casos podremos usar variables o constantes que podamos inicializar con un valor antes que necesites usarlas, sobre todo cuando queramos usar una variable o constante que abarque el ámbito global de una clase, como veremos más adelante. Su funcionamiento nos obligará a darles valor antes de ser inicializada la clase en sí misma lo que plantea problemas importantes.

De una manera u otra, la principal función de una opcional es poder permitir que una variable pueda tener valores vacíos o que un método de un objeto pueda devolver nada cuando no sea aplicable. En una clase podemos tener definidas propiedades o métodos que, por algún motivo, solo devuelvan valores válidos en determinadas circunstancias y en otros devuelvan un valor vacío. Para este tipo de casos existen las opcionales, entre otras posibles opciones que podamos pensar.

5 - SWIFT AVANZADO

5.4 Funciones Avanzadas

El uso de funciones, como hemos visto hasta ahora, es una de las cosas que mayor juego da dentro de Swift. No solo por las posibilidades de la llamada programación funcional, si no porque ellas mismas tienen muchas posibilidades: usar como variables, como parámetros de otra función, funciones anónimas como bloques o *closures*, uso de parámetros abreviados y sintaxis inferida… Pero la cosa no queda ahí porque las funciones tienen aun más posibilidades de uso, en un contexto más avanzado, y que vamos a repasar en este capítulo.

5.4.1 Funciones anidadas

Una función anidada o *nested function* es, básicamente, una función dentro de otra. Tal cual. Una de las cosas que Swift nos permite es que dentro del ámbito o el contexto de una función, declaremos otra a la que podemos llamar como si estuviéramos en un contexto diferente.

La forma más simple de estas funciones anidadas es aquella donde directamente ponemos una nueva función con toda su declaración dentro del ámbito de una función convencional.

```
func incrementar2(numero:Int) -> Int {

    func incrementar(num:Int) -> Int {
        return num + 2
    }

    return incrementar(numero)
}
```

En este caso, con esta implementación poco práctica, podemos ver en qué consiste una función anidada básica. Es importante tener en cuenta que la función `incrementar` es totalmente opaca fuera del ámbito de `incrementar2`, por lo que nadie sabe o puede acceder a ella (es totalmente privada). Además, la función anidada, `incrementar`, no puede acceder al ámbito de las variables de la función que la contiene, `incrementar2`, porque no comparten ámbitos de ejecución (si fuera un *closure* si podría).

Donde realmente son interesantes las funciones anidadas es cuando podemos enviar varios niveles de parámetros que preconfiguren la función principal para luego usar la función anidada dentro de esta en cada llamada. Algo así como una ejecución por pasos anidada de una o varias funciones.

Vamos a ver un ejemplo básico de una cuenta atrás.

```
func elegirCuenta(haciaatras: Bool) -> (Int) -> Int {

    func pasoAdelante(entrada: Int) -> Int { return entrada + 1 }
    func pasoAtras(entrada: Int) -> Int { return entrada - 1 }

    return haciaatras ? pasoAtras : pasoAdelante
}
```

En este caso lo importante es entender cómo funcionará esta función al tener dos parámetros de devolución. Si nos fijamos, el parámetro del centro está entre paréntesis (`Int`) mientras que el del final no los tiene. Eso significa que en los tres niveles de -> el primero de los parámetros (`haciaatras: Bool`) corresponde a la preconfiguración de la función, es decir a un parámetro que usaremos al asignar esta función a una variable. Más claramente: es el parámetro de inicialización de la función que dejará fijado este valor una vez lo asignemos a una variable.

```
var valor = -4
let contadoraCero = elegirCuenta(valor > 0)
```

En el momento que asignamos la función `elegirCuenta` a la constante `contadoraCero` hemos fijado de manera continúa el parámetro e inicializado la función anidada. Desde ese momento, internamente, la función se ejecutará como si el parámetro `haciaatras` ya estuviera definido a `return`.

Ahora podemos invocar a la función directamente a través de la variable, de forma que se ejecutará con el `valor` primero ya cargado.

```
while valor != 0 {
    valor = contadoraCero(valor)
}
```

Pero no todo es tan fácil porque el valor que hemos cargado como parámetro al inicializar `elegirCuenta` es `valor > 0` lo que significa que cada vez que llamemos a `contadoraCero` esta función volverá a evaluar la condición de manera implícita, aunque no lo indiquemos.

```
while valor != 0 {
    valor = contadoraCero(valor)
                -3
                -2
                -1
                 0
}
```

Como vemos, la cuenta hacia atrás se realiza sin problema porque primero hemos inicializado el primer parámetro y, a partir de ahí, cuando instanciamos `contadoraCero` lo que hace el sistema es llamar a la función `elegirCuenta` con el parámetro `haciaatras` con el valor precargado `valor > 0` (condición *booleana*) e inyectando un parámetro (`Int`) que será capturado por las funciones anidadas dentro como (`entrada: Int`). ¿Y cómo hace esto? Con la llamada que tenemos en el `return haciaatras ? pasoAtras : pasoAdelante` donde la llamada a las funciones `pasoAtras` o `pasoAdelante` infieren el parámetro inyectado directamente hacia ellas.

Si lo revisamos un par de veces veremos que es sencillo, aunque comprendo que estos son conceptos de lógica y funciones un poco complejos, que además la inferencia puede hacer algo confusos en un primer término. Lo que hay que entender es que la clave es el parámetro situado en el centro (`Int`) que indica que este es el dato a inferir o enviar a las funciones que estén anidadas dentro de nuestra función principal `elegirCuenta`.

5.4.2 Funciones genéricas para colecciones: map, filter, reduce y sorted.

Una de las funciones genéricas que mayor poder tienen en Swift y que implican un uso muy práctico de los *closures* y los parámetros abreviados de los mismos, son las funciones genéricas de colecciones que permiten realizar tareas muy pesadas de manera simple y en pocas instrucciones.

- `map` es un transformador que se ejecuta por cada elemento de la colección que se le pasa, aplicando una función o *closure* a cada caso y devolviendo el resultado de dicha aplicación de función en una nueva colección del mismo tipo y orden.
- `filter` es un condicionador que genera una nueva colección basada en el cumplimiento o no de un filtro que escoge o descarta valores de la colección.
- `reduce` es un acumulador que realiza una operación única que procesa todos los elementos de una colección para obtener un único resultado a través de aplicar una operación acumulada a cada caso (como un sumatorio).
- `sorted` se encarga de devolver una colección ordenada en función de un criterio que le permita comparar un valor con su siguiente y aplicarlo de manera recursiva a toda la colección con el fin de establecer el orden deseado.

La primera de ellas, `map`, tiene un potencial bastante importante en lo práctico si entendemos bien como funciona. Su cabecera genérica vendría a ser la siguiente:

```
func map(Array<U>(transform: (T) -> U) -> U[])
```

Es una función anidada que ejecuta un proceso prefijado en un parámetro y devuelve el resultado, es decir, ejecuta la función o *closure* que nosotros queramos (el que envíemos) en cada una de las ocurrencias de una colección, específicamente, un *array*.

```
let array = [5,3,10,9,3,8,12,15,1]
```

```
array.map( transform: Int -> U )
M  Array<U> map(transform: Int -> U)
Return an `Array` containing the results of calling `transform(x)` on
each element `x` of `self`
```

Si creamos un *array* cualquiera, veremos que la función `.map` forma parte de él como un método de una clase. En la llamada vemos que pertenece al *array* y que transforma el tipo `Int` de entrada (estamos usando un *array* de tipo `Int`) y devuelve una salida genérica. En la descripción lo vemos claramente: "*Devuelve un array que contiene los resultados de llamar a 'transform(x)' en cada elemento x de sí mismo*". Es decir, lo que hemos comentado. Nuestra responsabilidad por tanto es definir qué función o *closure* equivaldrá a *transform(x)* sabiendo que el valor que representa a cada valor independiente del *array* es el conocido argumento abreviado `$0`.

Hacemos la prueba más simple: que el *array* haga un `map` que se devuelva a sí mismo.

```
array.map({$0})
```

Como vemos lo único que hacemos es pasar una función que recibe un parámetro `Int` que en este caso es el valor en sí que es devuelto inmediatamente, pues no hemos indicado nada más.

Pero dentro de este *closure* podemos poner mucho más que nos permita transformar los valores del *array* en un solo paso. Por ejemplo, si queremos sumar 1 a todos los elementos es tan simple como esto:

```
array.map({$0+1})
```

Si vemos el resultado que nos devuelve el *playground* vemos claramente como el resultado de este `map` ha devuelto un *array* que ha sumado 1 a cada valor del mismo, simplemente pasándole como parámetro el *closure* `{$0+1}`.

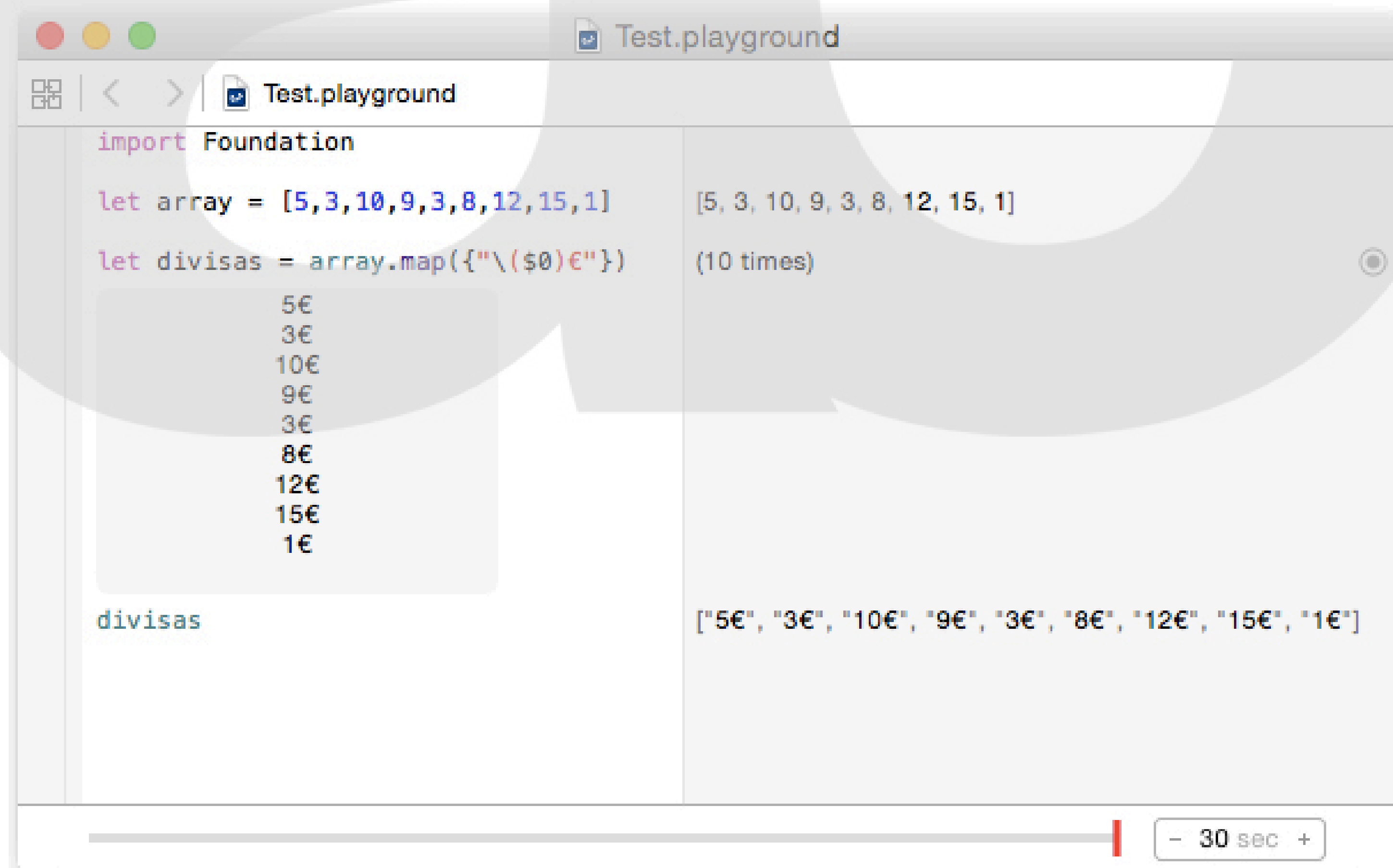
```
let array = [5,3,10,9,3,8,12,15,1]

array.map({$0+1})
        6
        4
        11
        10
        4
        9
        13
        16
        2
```

Pero tenemos que entender que `.map` es una función genérica que devuelve un resultado, por lo que no modifica en modo alguno el *array* original, ya que como ya sabemos un *array* es un tipo de dato por valor y no por referencia, de forma que se crea una nueva copia para ser modificada.

A partir de aquí, podemos exponer diferentes opciones que nos permitirían hacer esta función: por ejemplo, cambiar el tipo de dato con el que trabajamos. Imaginemos que queremos obtener un *array* de cadenas a partir de un *array* de cifras, porque queremos unir la divisa para poder procesar un listado.

```
let divisas = array.map({"\($0)€"})
```

Nuestro *closure* es tan simple como hacer una cadena que use la interpolación unido al símbolo del euro.



Automáticamente el nuevo *array* será de tipo cadena y habrá transformado, esencia de esta función, todos los valores del *array*.

Pero `map` nos permite también invocar funciones como parte del *closure* para cada elemento de un *array*. Podemos tener un *array* con multitud de elementos de un juego como `SKSpriteNode` que hemos ido definiendo y basta que hagamos:

```
nodos.map({scene.addChild($0)})
```

Igual podría pasar en una app donde tenemos una serie de objetos en nuestra vista y queremos aplicar un método a todos ellos, como por ejemplo aplicar un redibujado forzado en todos ellos con `setNeedsDisplay()`.

```
vista.subviews.map { $0.setNeedsDisplay() }
```

Lo que hay que tener muy claro es que `map` es una función de transformación que aplica lo que le digamos a todos los elementos del *array* uno a uno pero no convierte el *array* si no que devuelve una nueva copia.

Volvamos ahora al *array* de números que teníamos y vamos a intentar obtener uno nuevo cuya característica sea que solo contenga los números menores de 10 de todos los elementos incluidos. Lo normal, nuevamente, sería hacer una enumeración y una verificación, acumulando en un nuevo *array* cada elemento que cumpliera con la condición.

Pero también podemos usar la función `filter`. Una función cuya especificación coincide con la siguiente:

```
func filter(includeElement: (T) -> Bool) -> [T]
```

Normalmente para extraer los 6 elementos del *array* que son menores de 10 tendríamos que hacerlo con las siguientes líneas:

```
var nuevo = [Int]()
for num in array {
    if num < 10 {
        nuevo.append(num)
    }
}
```

Tras esto, `nuevo` contendría esta selección. Pero podemos hacerlo con `filter` y en una sola llamada.

```
var nuevo = array.filter({$0 < 10})
```

Lo que hace `filter` es aplicar una función llamada `includeElement` en cada elemento de un *array*. Para cada caso que devuelva *true*, pasará el filtro y lo incluirá en el nuevo *array* que devuelve.

```
var nuevo = array.filter( includeElement: Int -> Bool )
    M  Array<Int> filter(includeElement: Int -> Bool)

    Return an `Array` containing the elements `x` of `self` for which
    `includeElement(x)` is `true`
```

Lo realmente curioso de este caso, es que puede dar lugar a confusión porque cuando invocamos este método a través del *closure* que crea una condición que puede ser verdadera o no para extraer o no los datos del *array* a uno nuevo, si vemos el *playground* dirá que hemos ejecutado 10 veces, lo que implica que hemos pasado por todos los elementos del *array* origen lo que no quiere decir que el destino tenga los mismos. Si a continuación miramos los elementos que si están en `nuevo`, veremos que todo ha ido correctamente.

```
let array = [5,3,10,9,3,8,12,15,1]           [5, 3, 10, 9, 3, 8, 12, 15, 1]
var nuevo = array.filter({$0 < 10})          (10 times)
nuevo                                        [5, 3, 9, 3, 8, 1]
```

Con esta función conseguimos una simple instrucción de filtro de selección que es aplicable a cualquier condición que devuelva verdadero o falso. Si volvemos a nuestros ejemplos para una app, y pensamos en una vista llena de elementos diferentes pero queremos poder recuperar las etiquetas de toda la vista, no hay más que usar esta función.

```
let etiquetas = self.subviews.filter({$0 is UILabel})
```

Otra función a tener en cuenta es `reduce`, cuyo objetivo es reducir el total de elementos de un *array* a un único elemento, en función de un criterio que se proporcione desde un *closure*. Lo más natural de esta función es que la usemos como un sumatorio, como si fuera una hoja de cálculo donde queremos el total de una columna.

En esta ocasión la especificación genérica cambia un poco con respecto al resto. En `map` teníamos un único argumento de entrada y uno de salida, en `filter`, una única condición y también una salida, pero en esta ocasión es un poco más complejo porque a una única salida se le unen dos entradas. La primera es un valor inicializado que haga las veces de acumulador y la otra un *closure* que recibe dos parámetros de entrada y devuelve otra de salida, como sería una suma de dos números que devuelve uno. Una especificación que coincide con la mayoría de operadores aritméticos y que ya usamos anteriormente para montar nuestra calculadora.

En este caso, la especificación coincidiría con la siguiente llamada:

```
func reduce(initial: U, combine: (U, T) -> U) -> U
```

La construcción es mucho más sencilla gracias a la inferencia de sintaxis de los *closures* de forma que solo hay que dar un valor de acumulación o de inicio de las operaciones, seguido del *closure* de dos parámetros de entrada que devuelva un resultado, cuyo esquema coincide (como hemos dicho) con cualquier operador aritmético.

```
let total = array.reduce(0, combine: +)
```

Con esta sencilla instrucción hacemos un sumatorio de todos los valores que tiene el *array*. Si queremos verlo de una manera más sencilla, podemos pensar que estamos infiriendo los parámetros abreviados `$0` y `$1`, que también funcionarían en la llamada.

```
let total = array.reduce(0, combine: {$0 + $1})
```

De igual manera, podemos emplear una multiplicación pero al igual que sucedía con nuestra calculadora, el valor de inicio debería ser 1 para evitar una concatenación de multiplicaciones por 0 cuyo único resultado posible es: 0.

```
let total = array.reduce(1, combine: *)
```

Pero también podemos usar `reduce` para convertir entre tipos de datos y conseguir combinaciones como, por ejemplo, obtener todos los elementos numéricos de un *array* como una cadena separada por comas, tal vez porque queramos grabar en un fichero de propiedades estos valores y luego reconstruirlos en cualquier momento.

```
let total = array.reduce("", combine: { "\($0)" == "" ? "\($1)" : "\($0),\($1)" })
```

Cuando se procese el primer par de datos (el espacio en blanco y la primera cifra), el `reduce` usará solo la cifra obviando el espacio (para evitar un espacio de inicio) y a partir de ahí el *closure* procesará el resto de datos hasta el final. Con esto, obtenemos una serie de posibles combinaciones que nos permiten reducir el total de elementos de un *array* a un solo valor.

```
let array = [5,3,10,9,3,8,12,15,1]                    [5, 3, 10, 9, 3, 8, 12, 15, 1]
let total1 = array.reduce(0, combine: +)              66
    66
let total2 = array.reduce(1, combine: *)              5832000
    5832000
let total3 = array.reduce("", combine: { "\($0)" == "" ? "\
    ($1)" : "\($0),\($1)" })                          (10 times)
total3                                                "5,3,10,9,3,8,12,15,1"
    5,3,10,9,3,8,12,15,1
```

La última función que vamos a ver, es `sorted`, cuyo principal cometido es aplicar un criterio de ordenación a todos los elementos de un *array*.

La especificación de esta función también utiliza parámetros genéricos (como el resto de las funciones vistas, donde da igual que queramos usar cadenas o números de cualquier tipo) recibiendo, al igual que `reduce`, dos argumentos que, en este caso, deben tener una comparación binaria estableciendo su orden.

```
let orden =
    array.sorted( isOrderedBefore: (Int, Int) -> Bool )
M  Array<Int> sorted(isOrderedBefore: (Int, Int) -> Bool)

Return a copy of `self` that has been sorted according to
`isOrderedBefore`.  Requires: `isOrderedBefore` induces a `strict weak
ordering <http://en.wikipedia.org/wiki/ ...
```

El *closure* completo que habría que pasar sería con todos sus parámetros y devolviendo esta comparación que servirá de base de ordenación a la función para todo el *array*.

```
let orden = array.sorted({ (num1: Int, num2: Int) -> Bool in return num1 > num2 })
```

Pero como ya sabemos, ambos parámetros podemos inferirlos así como el tipo de dato que vamos a devolver (que siempre será un `Bool` porque el `return` lo lleva implícito) y dejar de usar `num1` y `num2`, con lo que nos queda la versión reducida con sus correspondientes argumentos reducidos.

```
let orden = array.sorted({$0 > $1})
```

Lo que estamos diciendo con este *closure* es que la lista se ordene de mayor a menor, partiendo del criterio que el primer número comparado sea mayor que el segundo en cada uno de los casos de la función.

```
let array = [5,3,10,9,3,8,12,15,1]        [5, 3, 10, 9, 3, 8, 12, 15, 1]

let orden = array.sorted({$0 > $1})       (26 times)
orden                                     [15, 12, 10, 9, 8, 5, 3, 3, 1]
```

Y si observamos el resultado veremos que curiosamente, se nos indican las vueltas que ha tenido que dar el algoritmo, recursivamente, para ordenar correctamente la lista.

Antes de cerrar, no podemos dejar pasar la oportunidad de ver cómo funcionarían en conjunción todos estos argumentos, porque siempre podemos usar combinadas estas opciones. Por ejemplo, puede que queramos un listado como cadena hecho con `reduce` pero del *array* ordenado, lo que nos dejaría una instrucción así.

```
let orden = array.sorted({$0 > $1}).map({"\($0)€"}).reduce("", combine: { "\($0)" == "" ? "\($1)" : "\($0),\($1)" })
```

Como `sorted` devuelve un *array* podemos unirlo a cualquier otro de los métodos. En este caso hacemos una triple operación: primero ordenamos de mayor a menor con `sorted` y al *array* resultante le hacemos un `map` que le añade el símbolo del € a cada número de cada posición, ya convertido en cadena. Por último, hacemos el `reduce` para obtener una sola cadena que tenga los números con sus símbolos separados por comas, y listo.

Como puede verse, la versatilidad y funcionalidad de estas funciones combinadas con los *closures* es bastante alta.

5.4.3 Funciones parcializadas

Las funciones *curried* o parcializadas, deben su nombre al matemático lógico estadounidense Haskell Curry, creador en 1990 del lenguaje de programación funcional que lleva su nombre: el Haskell. Este concepto, inexistente en Objective-C, nos plantea un paradigma de uso funcional completamente nuevo basado en la invocación por partes de una función.

En su concepción, una función currificada o parcializada es aquella donde parte de sus parámetros pueden ser aplicados anteriormente a la llamada de la función, por lo que obtenemos una ejecución diferida de una función o una pre-instanciación con parte de sus parámetros definidos. Una forma más simple de entenderlo es pensar en las funciones currificadas como funciones donde se establecen unos valores por defecto que no son definidos en el momento de crear la función, sino previamente a invocar dicha función cuando almacenamos la misma en una variable que hará las veces de llamada a la misma.

Más simple aun: una función currificada es aquella que contando con n grupos de parámetros, no será ejecutada hasta que todos ellos tengan valor, pudiendo enviar los parámetros en diferentes fases con lo que podemos precargar parámetros.

La forma de usar la currificación es poniendo diferentes grupos de parámetros dentro de paréntesis, donde el primero será el que se tome por defecto y el último el referido. Vamos a crear una función que sume dos números. Normalmente haríamos esto:

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

De esta forma, cuando llamemos a `sumar` hemos de hacerlo enviando ambos parámetros a la vez para obtener el resultado, como por ejemplo:

```
let suma = sumar(2, 3)
```

La variable `suma` tomaría el valor 5. Pero ahora vamos a definir `sumar` como una función currificada y vamos a separar ambos parámetros:

```
func sumar(x:Int)(y:Int) -> Int {
    return x + y
}
```

Lo que conseguimos con esto es poder invocar dentro de una variable el estado precargado de una función, como instanciar un método de una clase con un parámetro pero que sigue siendo función.

```
let suma = sumar(2)
```

Ahora tenemos en la variable `suma` una función. No un valor, un entero, una clase o cualquier otro tipo. Lo que tenemos es la referencia a una función cuya declaración es:

```
let suma: (y: Int) -> Int
```

El valor de x (un 2) se ha precargado, de forma que la función `suma` que recoge un `Int` y devuelve otro como resultado estaría construida internamente de la siguiente forma:

```
func suma(y:Int) -> Int {
    return 2 + y
}
```

Esta función que vemos es la que ahora mismo está almacenada dentro de `suma`. Ahora invocamos `suma` con el valor 3 y nos sumará este al 2, obteniendo un 5.

```
suma(y: 3)
```

Una forma muy útil y sencilla para hacer multitud de cosas pues podemos crear funciones con parámetros dinámicos definidos en tiempo de programación.

Vamos a ver otro ejemplo. Supongamos que queremos una función que nos permita multiplicar por un valor x otro valor a. Para ello creamos la función currificada de una forma similar a la que ya hemos creado, pero esta vez multiplicando.

```
func multiplicaPor(x:Int)(a:Int) -> Int {
    return x * a
}
```

Ya tenemos nuestra función y vamos a pensar que queremos multiplicar por 3 el número que se nos pase. Vamos entonces a crear una instanciación de la función dentro de una variable con el valor 3 para la x.

```
let multiplica3 = multiplicaPor(3)
```

Ahora, podíamos llamar a `multiplica3` con cualquier número y nos lo multiplicaría por 3. Pero, ¿y si tenemos un *array* de números? Pues simplemente invocamos el mapeado de valores del *array* enviando como función de transformación genérica `multiplica3`, y tendríamos mágicamente todos los valores del *array* multiplicados por 3 en una sola instrucción.

```
let numeros = Array(1...10)
numeros.map(multiplica3)
```

Debemos tener presente que hablamos de valores, no de referencias, por lo que esta instrucción nos devuelve los valores *array* transformados pero no transforma el propio *array* ni sus valores. Si vemos el contenido de `numeros`, este permanece inmutable (por razones obvias además, porque es `let`) . Tendríamos que crear una nueva variable para almacenar el resultado de la transformación al aplicar nuestra función currificada.

```
let numeros3 = numeros.map(multiplica3)
```

La currificación está muy embebida dentro del lenguaje, de forma que incluso las llamadas a métodos de una clase están parcializadas por defecto.

Vamos a ver una versión reducida de la clase `Personaje` para entender cómo funciona el sistema y cómo se instancian los métodos de manera interna en todo el lenguaje a través de la currificación.

```
class Personaje {
    var vida:Int = 100

    func quitarVida(cantidad: Int) {
        vida -= cantidad
    }
}
```

Está claro que si instanciamos un objeto a partir de esta clase, la forma que quitarle vida es invocando su método como hemos visto anteriormente.

```
let elfa = Personaje()
elfa.quitarVida(50)
```

La forma convencional en que funcionamos siempre con métodos a la hora de enviar mensajes está clara. Pero, ¿cómo funciona internamente el sistema? Cuando nosotros creamos un método de una clase, en realidad dicho método se guarda en una variable interna de manera genérica, sin depender de instanciación alguna.

```
let quitarVida = Personaje.quitarVida
```

Si hacemos esta asignación, podemos comprobar que el *playground* no se queja en modo alguno y automáticamente lo que tenemos en la constante `quitarVida` es una función. A pesar de no haber indicado ningún parámetro, el lenguaje ha aceptado esta forma y ha dejado almacenada la llamada a la función, pero ojo, sin pertenecer a instanciación alguna si no solo el método de la propia clase a su nivel de definición.

Este procedimiento es el que hace el propio lenguaje internamente con todos los métodos de una clase. Simplemente tenemos que empezar a escribir su nombre para que la ayuda nos muestre su tipificación y veamos cómo funciona esta función que tenemos en la variable `quitarVida`.

Es de un tipo `Personaje -> (Int) -> ()`, indicando que tiene tres parámetros con un parámetro intermedio, muy parecido a como vimos también que funcionaban algunos casos de funciones anidadas. Esto significa que espera que le pasemos primero un objeto de la clase `Personaje`, y luego un número a lo que devolverá una instancia vacía como resultado.

Hagámoslo entonces y llamemos a esta función dentro de la variable pasándole los dos parámetros que espera: el objeto de la clase `Personaje` y el número de tipo `Int`. Ambos en grupos separados de paréntesis.

```
quitarVida(elfa)(20)
```

Aparentemente no ha hecho nada (porque recordemos que devuelve una instancia vacía). Pero vamos a comprobar la propiedad **vida** de nuestro `elfa`.

```
let elfa = Personaje()                          {vida 100}
elfa.quitarVida(50)                             {vida 50}

let quitarVida = Personaje.quitarVida           (Function)

quitarVida(elfa)(20)
elfa                                            {vida 30}
```

Se ha reducido en `20`, el número que hemos pasado. Es decir, hemos invocado el método `quitarVida` de `Personaje` que pertenece al objeto `elfa`, directamente desde la primitiva de la función guardada en una constante, pasándole como parámetros el objeto y el parámetro que espera el propio método. Dicho de otro modo: el método es independiente y lo que hacemos es invocarlo pasándole el objeto de la clase a la que pertenece el método y los parámetros.

Esta es la forma en que el sistema funciona internamente, lo que nos puede dar una mejor idea de cómo funcionan las funciones currificadas y la importancia que tienen en el propio funcionamiento del lenguaje y su fundación.