ACERCA DEL AUTOR

Enrique Blasco Blanquer (Alcoy [Alicante], 1986) es ingeniero técnico en Informática de Gestión por la Politécnica de Valencia (campus de Alcoy EPSA). En 2005 estudió el Ciclo Formativo de Grado Medio en Explotación de Sistemas Informáticos; en 2007, el Grado Superior en Desarrollo de Aplicaciones en CIP FP Batoi. En 2009 empezó en la EPSA UPV la Ingeniería Técnica en Informática de Gestión. Finalmente, en 2011 realizó el Máster en Aplicaciones Móviles Android e iOS en la escuela de nuevas tecnologías CICE de Madrid.

Durante un año estuvo desarrollando aplicaciones web en OSOLE Alcoy, trabajó en la creación de las aplicaciones para iPad de Room Mate Hoteles y Husa Hoteles para la empresa Concept TYC de Madrid. En 2012 empezó su carrera profesional como profesor en la escuela de nuevas tecnologías CICE en Madrid. Impartió durante cuatro años los cursos de web con HTML, CSS3, JavaScript, jQuery, PHP y MySQL, el curso certificado de ORACLE con Java SE y Java EE, así como el curso de desarrollo de aplicaciones móviles para iOS y Android, además del de diseño e implementación de aplicaciones hibridas utilizando Cordova PhoneGap. Al mismo tiempo trabaja como desarrollador en distintas aplicaciones para iOS y Android. Actualmente vive y trabaja en Valencia como desarrollador de sistemas SAP FRIORI utilizando la tecnología SAPUI5 en la empresa SEIDOR Valencia.

INTRODUCCIÓN

Dicen que la tecnología tiene vida propia. Nace, crece, aprende, envejece, muere y deja paso a tecnologías preparadas para el nuevo entorno; así sobrevive y sigue avanzando. Por eso somos responsables de seguir su camino estudiando los nuevos cambios y las nuevas necesidades del entorno. En esto, Apple es un experto.

Sus productos estrella —iPhone, iPad y ahora Watch— están en constante evolución. Cada año cambian tanto sus dispositivos como su sistema operativo y, por tanto, la forma en que se programa.

En el 2014 fue presentado un nuevo lenguaje de programación llamado Swift, un lenguaje seguro de desarrollo rápido y fácil de entender, creado por Apple enfocado al desarrollo de aplicaciones para iOS y MAC OS X.

Apple cambió nuestra forma de pensar y ver los dispositivos móviles, nadie, ni en las mejores películas de ciencia ficción, podría haber imaginado cómo terminarían influenciando nuestras vidas estas máquinas. Parte del éxito ha sido encontrar un gran equilibrio entre interfaz, diseño y rendimiento. Aplicaciones fáciles de usar e intuitivas, adaptadas al uso en constante movimiento, y una gran campaña publicitaria, han conseguido que parte de nuestras vidas se vea cambiada.

Los nuevos dispositivos de Apple están llenos de grandes posibilidades: GPS, mapas, cámara de fotos, sensores de movimiento, de aceleración… Con todo esto, una buena idea e imaginación podemos llegar a crear grandes aplicaciones. Pequeñas obras de arte para el uso diario, el entretenimiento o simplemente por la necesidad de crear. Por eso la programación, en mi opinión, es tan bonita, porque permite crear algo desde cero.

Pero para poder crear, primero tenemos que aprender. Este es el primer paso.

SOBRE EL LIBRO

Este libro está enfocado a que aprendas las herramientas básicas para el desarrollo de aplicaciones para iOS. Ejemplos prácticos que van evolucionando desde conceptos mínimos del lenguaje Swift hasta aplicaciones con conexión a base de datos, autenticación de usuarios y despliegue en la App Store.

Paso a paso se van introduciendo la gran mayoría de conceptos necesarios para poder tener prácticas con código simple y limpio. Al final del libro dispondrás de una gran variedad de pequeñas prácticas que juntas componen todo lo necesario para una gran aplicación terminada.

De forma visual y práctica aprenderás a desarrollar tus propias aplicaciones para empezar esta gran aventura del desarrollo. Pero gran parte del trabajo corre de tu cuenta. Tener motivación e ilusión, además de tiempo para realizar cada una de las prácticas, es esencial para el buen aprendizaje.

En cada tema del libro se explica el concepto teórico que se va a desarrollar, seguido de ejemplos prácticos para afianzarlo. Al final de cada práctica se sugiere un conjunto de retos o ampliaciones; es recomendable realizar alguno de ellos para aprender más sobre lo expuesto en las prácticas.

En la programación no se aprende hasta que uno se sienta a desarrollar sus propios proyectos. Por eso se necesita un tiempo para el desarrollo personal, un espacio agradable y tranquilo fuera de distracciones y redes sociales.

En el primer tema aprenderemos conceptos básicos sobre el lenguaje de programación Swift. Sintaxis y herramientas necesarias para el correcto aprendizaje en el desarrollo de aplicaciones en iOS.

En el tema 2 crearemos nuestro primer proyecto, aprendiendo el entorno gráfico de Xcode, la estructura de un programa, elementos básicos para la interfaz del usuario; y nos empezaremos a sentir cómodos experimentando la calidad de programar para iOS. También conoceremos el protocolo modelo-vista-controlador, muy importante para entender el funcionamiento y organización de un proyecto.

En el tema 3 aprendemos todas las herramientas para la creación de las vistas de nuestras aplicaciones y la navegación entre ellas. Con este tema podremos desarrollar nuestra primera aplicación completa con datos estáticos, es decir, textos planos introducidos por el desarrollador.

El tema 4 está enfocado a la consistencia de datos de forma local usando los conocimientos obtenidos hasta el momento. Empezaremos a desarrollar aplicaciones donde introducir datos, eliminar, listar y editar de forma permanente en nuestro dispositivo. Crearemos una aplicación completa para la gestión de notas personales.

En el tema 5, uno de los temas más complejos, entramos en el fascinante mundo del desarrollo de aplicaciones con datos en un servidor externo. Aprenderemos a montar nuestro propio *webservice* con PHP y a interactuar con nuestra aplicación en iOS por medio de XML para la creación, listado, eliminación y edición de datos de forma remota. Para afianzar todos los conceptos desarrollaremos una aplicación donde podremos enviar mensajes entre usuarios. Una vez llegados a este punto ya estaremos preparados para crear un gran abanico de aplicaciones.

En el tema 6 empezamos a exprimir las posibilidades de los dispositivos de Apple. Accederemos a la cámara y a nuestra biblioteca de fotos. También aprenderemos a reproducir audio y vídeo con Swift.

En el tema 7 entramos en el mundo de los mapas, aprenderemos a insertar un mapa en nuestras aplicaciones, a obtener la posición del dispositivo y a jugar con las grandes posibilidades que nos ofrece este mundo.

En el último tema aprenderemos a subir nuestras aplicaciones a la App Store para así poder compartir con todo el mundo nuestras pequeñas obras de arte.

Por último, quiero destacar que el mundo del desarrollo de aplicaciones es interminable, por lo que aquí no acaba todo, ahora toca desarrollar una aplicación para una necesidad real. Solo así, desarrollando, no dejaremos nunca de aprender y mejorar.

CONOCIMIENTOS PREVIOS

No hace falta ser un experto para desarrollar aplicaciones. Tener conocimientos previos de programación ayuda al mejor entendimiento y a un aprendizaje más rápido, pero no es necesario. Este es un libro práctico y visual con conceptos pensados para todos los públicos.

Quienes tengan conocimientos de programación encontrarán un libro donde aprender de forma rápida y clara los puntos fundamentales de toda aplicación en iOS.

Aquellos que ya desarrollen para iOS, pero con Objective-C, podrán tener una gran ayuda y un manual para encontrar las grandes diferencias en el desarrollo con Swift.

Si es la primera vez que te adentras en el mundo de la programación, este será tu primer paso en un gran mundo. Aquí se busca potenciar la motivación del desarrollador, al mostrarle sus pequeños programas en funcionamiento.

Pedimos y damos por supuesta la motivación del desarrollador, así como su dedicación.

REQUISITOS DEL SISTEMA

Para desarrollar las prácticas de este libro es necesario disponer de un Mac de Apple con OS X o superior, y tener instalado el entorno de desarrollo Xcode 7, ya que viene preparado para el lenguaje Swift.

No hace falta una gran máquina; en mi caso, todas las prácticas y el desarrollo se hacen sobre un MacBook Air de once pulgadas, y el rendimiento es más que suficiente:

Es recomendable que el sistema esté actualizado, tanto el OS como el Xcode.

Las nuevas versiones traen pequeños cambios en la sintaxis del lenguaje, o nuevas funciones. Durante el tiempo de desarrollo puedes encontrarte con una actualización, lo cual obligará a introducir pequeñas variaciones en el código para que este se adapte perfectamente.

El código presentado en este libro funciona bajo Xcode 7.2.

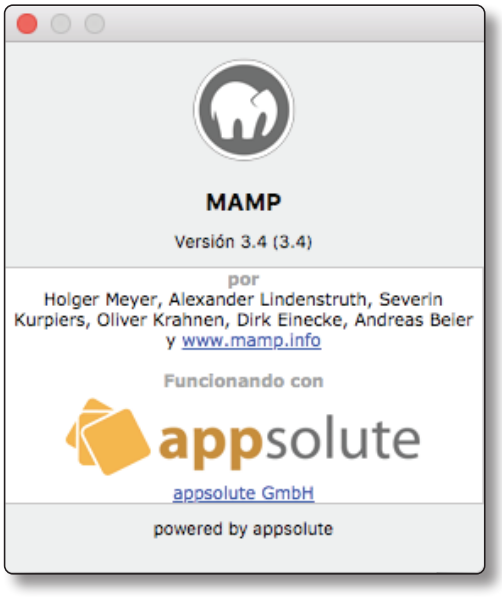
También es recomendable, pero no obligatorio, contar con un dispositivo iPhone o iPad actualizado a la última versión del sistema operativo, para poder probar las aplicaciones. En Xcode viene integrado un simulador de dispositivos, pero la experiencia final debe probarse sobre un dispositivo físico. Además, probar las aplicaciones físicamente motiva a la hora del desarrollo, ya que ves en funcionamiento tus creaciones.

En el capítulo de base de datos con MySQL desarrollamos un pequeño *webservice* con PHP utilizando el entorno de desarrollo NetBeans 8.0.2 y MAMP para el servidor local.

NetBeans es un entorno de desarrollo para aplicaciones Java SE, Java EE, web y PHP entre otros. Es gratuito y funciona bajo plataforma MAC OS X. Llegado el momento explicaremos de dónde descargar la máquina virtual de Java, y los posibles problemas con ella.

En este caso no tienes que usar NetBeans de forma obligada, dejo a tu elección el entorno de desarrollo para PHP.

Para crear un servidor local y así probar nuestras aplicaciones conectando a un *webservice* utilizamos MAMP para OS X, ya que es fácil de usar y bastante estable.



Al igual que con NetBeans, explicaremos en su momento dónde obtener el programa y su funcionamiento.

Existen otros programas, como XAMP, que realizan las mismas funciones, por lo que es de tu elección el uso del programa para lanzar un servidor local con base de datos MySQL.

Por último, necesitas tener una cuenta de Apple. En el siguiente tema vamos a explicar cómo crear una.

CREAR UNA CUENTA DE APPLE

La cuenta de Apple hace falta para descargar el entorno de desarrollo Xcode y probar nuestras aplicaciones en un dispositivo físico.

Es gratuito y muy fácil de hacer. Vamos a ver los pasos para obtener una cuenta (si ya tienes una ID de Apple, puedes saltarte estos pasos):

1. Accede por medio de tu navegador a la página *https://appleid.apple.com/* y entra en **Crear tu ID de Apple**.

2. Introduce todos los campos, como el nombre, apellidos, correo, etc. y presiona sobre **Continuar** (introduce un correo válido, ya que te enviarán un código de confirmación).

3. Accede a la dirección de correo que has puesto en el paso anterior y obtén la clave de confirmación.

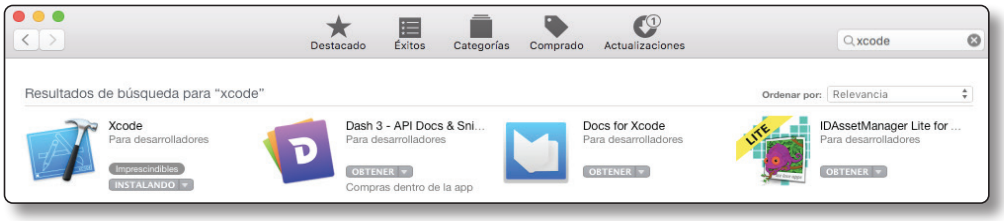
4. Introduce la clave de confirmación y ya tendrás tu cuenta de Apple preparada.

PREPARAR EL ENTORNO DE DESARROLLO

Xcode es el entorno de desarrollo integrado de Apple. Incluye la colección de compiladores del proyecto GNU, y puede compilar código C, C++, Swift y Objective-C mediante una amplia gama de modelos de programación. Es la herramienta necesaria para el desarrollo de aplicaciones con iOS.

Antes de empezar lo tenemos que instalar de la App Store. El único requisito es tener instalada la última versión del OS de nuestro Mac.

1. Accedemos a la App Store y en el buscador introducimos “xcode”.



2. Buscamos el programa en el resultado de búsquedas y presionamos sobre **Instalar**. El proceso depende de la conexión a Internet y puede llegar a tardar más de una hora en su descarga completa e instalación.

3. La instalación es automática, una vez descargado se procederá a la instalación del programa.

4. Una vez instalado, ya lo podemos encontrar en nuestra carpeta de aplicaciones. Si lo ejecutamos nos encontraremos con la pantalla de bienvenida:

Aquí podremos abrir proyectos existentes, crear un nuevo Playground, crear un nuevo proyecto o abrir un proyecto existente en un repositorio SCM.

Puedes ocultar esta ventana de bienvenida quitando el selector donde indica que se muestre al lanzar el Xcode.

Llegados a este punto, ya tenemos todo preparado para empezar con el desarrollo de aplicaciones iOS con Swift. ¡Buena suerte!

1

SWIFT

Swift es un nuevo lenguaje de programación para iOS, OS X, watchOS y aplicaciones tvOS basado en lo mejor de C y Objective-C, sin las limitaciones de compatibilidad de C. Añade patrones de programación segura y características modernas para que la programación sea más fácil, flexible y divertida. Con Swift han querido hacer borrón y cuenta nueva para poder re-imaginar el desarrollo de software.

Swift resulta familiar para los desarrolladores de Objective-C, no ha cambiado tanto la forma en que se desarrollan las aplicaciones, lo cual facilita el cambio del nuevo lenguaje. Adopta la legibilidad de los patrones con nombre de Objective-C y el poder del modelo de objetos dinámicos, esto posibilita combinar Swift con el código Objective-C. Con Swift se introducen muchas nuevas características y se unifican las partes procesales y de orientado a objetos del lenguaje.

Para los nuevos desarrolladores, Swift es muy amigable. Es el primer lenguaje de programación para software de altas prestaciones tan expresivo y agradable como un lenguaje de *script* sencillo. Es compatible con el Playground, una característica innovadora que permite a los programadores experimentar con el código y ver los resultados inmediatamente sin la sobrecarga del desarrollo de una aplicación.

En mi opinión, Swift ha conseguido ser un lenguaje de programación moderno, preparado para las nuevas generaciones y muy amigable, que consigue que nos centremos en la lógica de la aplicación y no en el lenguaje en sí. Es como disfrutar de conducir sin pensar en cómo se conduce el coche. Todo esto mezclado con la sabiduría de la cultura más amplia de la ingeniería Apple.

El compilador está optimizado para el rendimiento y el lenguaje para el desarrollo, sin comprometer a ambos. Por eso Swift es una inversión de futuro para todos los desarrolladores que queramos vivir en el mundo de Apple.

Tenemos que pensar que Swift es un "recién nacido", pero con la experiencia del más veterano. Por ello, nos vamos a encontrar con un lenguaje en constante evolución en las nuevas características y capacidades.

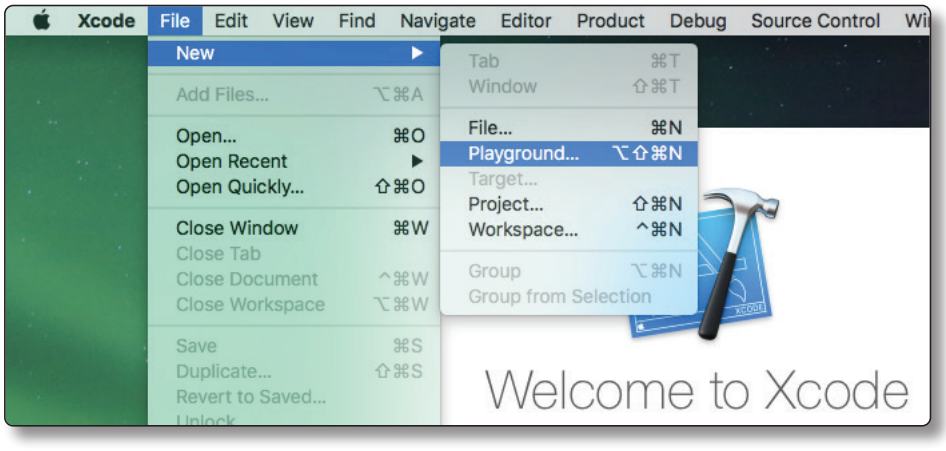
1.1 PLAYGROUND

Para empezar a conocer el nuevo lenguaje de Apple, Swift, tenemos que conocer "el parque infantil".

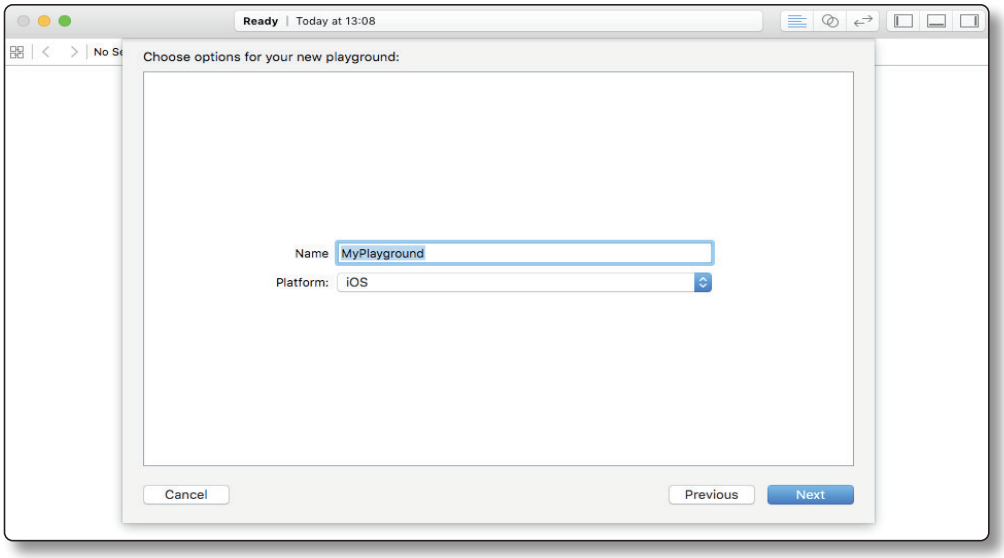
El Playground (parque infantil) es un entorno de codificación Swift interactivo que evalúa cada declaración y muestra los resultados cuando se realizan cambios sin la necesidad de crear un proyecto. Esto facilita mucho el aprendizaje del lenguaje, así como la puesta en práctica de pequeños algoritmos que quieras desarrollar para tu proyecto real. Se utiliza para aprender y explorar Swift, como piezas prototipo para una aplicación y para crear ambientes de aprendizaje. El entorno interactivo de Swift te permite experimentar con algoritmos, explorar las API del sistema e incluso crear vistas personalizadas.

Es, por todo esto, un entorno perfecto para empezar a experimentar con este fascinante lenguaje. ¡Manos a la obra!

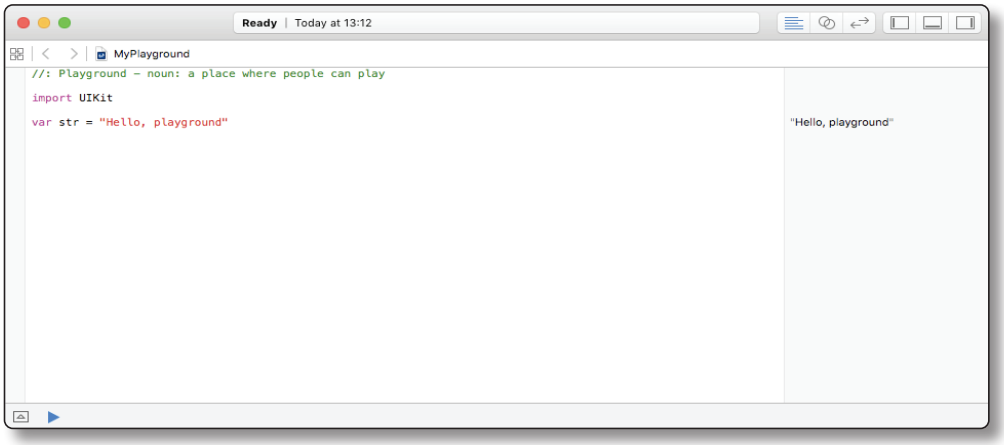
En primer lugar vamos a ejecutar Xcode, donde nos saldrá nuestra ventana de bienvenida. De todas las opciones seleccionaremos **Get started with Playground**. En caso de que no aparezca la ventana de bienvenida podremos crear un nuevo Playground accediendo al menú de herramientas de Xcode, **File → New → Playground**.



A continuación nos saldrá una ventana donde introducir el nombre de nuestro fichero Playground. En **Plataforma** podemos seleccionar para qué sistema operativo queremos experimentar, en nuestro caso iOS:



Presionamos sobre el botón **Next**, indicamos una ubicación para guardar el fichero y ya tenemos nuestro Playground preparado para estudiar:



El Playground es como un bloc de notas moderno muy sencillo de usar. Tenemos en la parte de la izquierda, la pizarra, donde insertar nuestro código; en la derecha vemos línea por línea el resultado de dicho código cada vez que detecte un cambio.

Al iniciar un nuevo Playground nos encontramos con el siguiente código:

```
//: Playground - noun: a place where people can play

import UIKit

var str = "Hello, playground"
```

Tenemos un comentario con las dos barras //, sentencia de código que el compilador no interpreta. A continuación tenemos un **import UIKit**, esto añade a nuestro Playground un conjunto de funciones con respecto a la interfaz del usuario. Esta librería nos permitirá experimentar con elementos de la interfaz del usuario y aprender un poco más de ellas.

Finalmente nos encontramos con la definición de una variable llamada **str** con el valor **Hello, playground**.

Todos estos elementos los estudiaremos más adelante. De momento lo importante es notar que, junto a la línea de la variable, podemos ver, en la derecha de la pantalla, su contenido.

Con todo esto, vamos a empezar a ver los conceptos básicos del lenguaje.

1.2 BÁSICOS

Swift proporciona sus propias versiones de todos los tipos C y de Objective-C fundamentales, incluyendo **Int** para enteros, **doubles** y **flotas** para valores en punto flotante, **Bool** para valores booleanos y **String** para los datos textuales.

Al igual que C, Swift utiliza variables para almacenar y hacer referencia a valores con un nombre de identificación; así como las constantes, variables cuyos valores no se pueden cambiar, que son mucho más potentes que las constantes en C.

Swift introduce tipos avanzados que no se encuentran en Objective-C, como las tuplas, elementos que te permiten crear y pasar una agrupación de valores. Se pueden usar para devolver varios valores de una función como si fueran un único valor.

Una parte un poco más complicada de entender son los valores opcionales, algo novedoso en Swift; son comparables a los valores nulos en Objective-C. Son los valores encargados de ocupar la ausencia de un valor en una variable no solo de una clase sino de cualquier tipo. Son más seguros que los punteros nulos de Objective-C.

1.2.1 Comentarios

Usamos los comentarios para incluir texto no ejecutable en el código, como una nota o recordatorio. Los comentarios son ignorados por el compilador Swift cuando se compila el código.

Los comentarios en Swift son muy similares a los de cualquier otro lenguaje de programación, para emplearlos hay que introducir la doble barra inclinada al principio de una línea (//) y comentar la sentencia. En caso de más de una línea utilizamos /* al principio de la sentencia y terminamos con */.

```
//Experimentando con el lenguaje Swift

/*
    Comentarios con muchas líneas
    Podemos escribir un texto explicativo o recordatorio
*/
```

1.2.2 Constantes y variables

Las constantes y variables son espacios en memoria de la máquina con un nombre asociado a un valor de un tipo particular. Las constantes son variables que no pueden cambiar su valor una vez está establecido, mientras que en una variable se puede establecer un valor diferente en cualquier momento.

Para poder usar las constantes y variables, antes se tienen que declarar, proceso muy sencillo en Swift. Para declarar una variable constante tenemos que usar la palabra reservada **let** seguida del nombre de la constante y un valor. Para el caso de las variables se utiliza la palabra reservada **var**. Vamos a ver un ejemplo. Introducimos en nuestro Playground las siguientes sentencias de código:

El código se lee de la siguiente forma:

```
var numDeMascotas = 2 //variable
let numPI = 3.1416 //constante
```

Hemos creado una variable llamada **numDeMascotas** con el valor asociado 2. Después hemos declarado una constante llamada **numPI** con el valor 3.1416.

Antes de seguir vamos a prestar atención a varias cosas en la declaración de las variables.

1. Los nombres que pongamos a nuestras variables deben empezar por minúsculas sin acentos y sin espacios en blanco.

2. No hace falta terminar la sentencia con un punto y coma.

3. La notación correcta para definir valores es con un símbolo igual (=).

4. Los números se introducen sin comillas. El punto es el separador de los decimales.

5. Las cadenas o valores alfanuméricos han de ir entre comillas dobles; si queremos definir un carácter, podemos poner el valor entre comillas simples.

También podemos declarar múltiples constantes y variables en una línea, separadas por comillas:

```
var x = 29.0, y = 1.72, z = 65.7
```

En Swift se hace mucho hincapié en usar constantes si el valor de la variable no va a modificar su valor. Las variables se utilizan solo para almacenar valores que deben ser capaces de cambiar. A lo largo de los proyectos, el compilador marcará aquellas variables que no cambian en su valor, sugiriendo que lo modifiques con la declaración **let**, constante. Ya que de esta forma optimizamos mucho mejor nuestras aplicaciones en cuanto a memoria y rendimiento.

La función print()

Finalmente, para poder pintar una variable o constante utilizamos la función **print(nombreVariable)**:

```
print(numDeMascotas) //pintando 2
```

La función **print** permite imprimir uno o varios valores por la consola del compilador. Por defecto, la función termina la línea que se imprime con un salto de línea (**/n**).

Con Swift podemos usar interpolación de cadenas para incluir el nombre de una constante o variable como un marcador de posición de una cadena más larga envolviendo la variable o constante entre paréntesis y con una barra invertida al principio:

```
print("El número de mascotas es: \(numDeMascotas)")
```

Esta sentencia pintará “El número de mascotas es: 2”.

1.2.3 Tipos de anotaciones

En la declaración de las variables anteriores podemos observar que no hemos indicado el tipo de variable que podemos almacenar. Esto nos facilita a la hora de declarar una variable de la que no sabemos el tipo de valor, pero en ocasiones necesitamos anotar qué tipo de valor puede almacenar para conseguir una seguridad en nuestro programa. En Swift, como en otros lenguajes de programación, tenemos lo **String**, **int** y **dobule**, **Boolean** para los diferentes valores de una variable.

Para poder definir una variable o constante indicando el tipo se introducen dos puntos después del nombre seguido de un espacio en blanco y del nombre del tipo que se va a usar.

Vamos a declarar una variable llamada **nombre** que puede almacenar valores de tipo **String**, es decir, un valor textual:

```
var nombre : String
```

Ahora la variable nombre puede contener cualquier valor de tipo cadena sin error:

```
nombre = "Enrique"
```

También podemos definir múltiples variables del mismo tipo en una línea de código, separados por comas con el tipo de anotación al final de los nombres de las variables:

```
var peso, altura: Double
```

1.2.4 Números enteros

Las variables y constantes de tipo enteros pueden almacenar valores numéricos no decimales positivos, cero o negativos.

```
let min = UInt8.min //igual a 0
let max = UInt8.max //igual a 255
```

Con Swift podemos asignar números enteros de 8, 16, 32 y 64 bits.

En muchas ocasiones no queremos especificar el tamaño del número entero. Swift ofrece un tipo entero adicional, el **Int**:

```
let numTransportes: Int = 6765
```

64 bits obtiene el tamaño de un **Int64**. A menos que se necesite trabajar con un tamaño específico de número entero, se recomienda utilizar el **Int** para valores enteros. Esto ayuda a la consistencia del código, incluso en plataformas de 32 bits. **Int** puede almacenar cualquier valor comprendido entre -2147483648 y 2147483647, un rango lo suficientemente grande para almacenar una gran cantidad de números enteros.

1.2.5 Números de punto flotante

Los números de punto flotante son números con decimales, positivos y negativos.

Los tipos de punto flotante pueden representar una gama mucho más amplia de valores que los tipos enteros. En Swift existen dos tipos de número con punto flotante:

1. **Double**: Representa un número de 64 bits con una precisión de unos 15 dígitos decimales.

2. **Float**: Representa un número de 32 bits con una precisión de unos 6 dígitos decimales.

El tipo de punto flotante apropiado depende de la naturaleza y el rango de valores que necesita para trabajar en el código. En situaciones en las que cualquiera de los tipos sería apropiado se prefiere el **Double**:

```
let peso: Double = 65.5678
let altura: Float = 1.765
```

1.2.6 Booleans

En Swift existe un tipo booleano básico llamado **Bool** cuyo valor puede contener **true** o **false**:

```
var elCieloEsAzul : Bool = true
var elSolEsAmarillo : Bool = false
```

Los valores booleanos son particularmente útiles cuando se trabaja con sentencias condicionales tales como la sentencia **if**:

```
if elCieloEsAzul {
    print("El cielo es azul")
```

```
}else{
    print("El cielo NO es azul")
}
```

1.2.7 Tuplas

Las tuplas permiten contener múltiples valores en un único valor compuesto. Los valores dentro de una tupla pueden ser de cualquier tipo y diferentes entre sí:

```
var http404Error = (404,"Not Found")
```

En este ejemplo vemos una tupla con un entero y una cadena de texto separados con una coma y entre paréntesis. Se pueden crear tuplas de cualquier permutación de tipos, y pueden contener tantos tipos diferentes como se desee.

A la hora de declarar una tupla, no solo ha creado nuestra variable con valores múltiples, sino que les ha asignado un índice a cada uno de manera automática empezando por la posición 0.

En este caso ha asignado el valor 0 para el entero y el valor 1 para el texto, es decir, en caso de que queramos pintar los valores de la tupla, introduciremos lo siguiente:

```
print(http404Error.0) //pinta 404
print(http404Error.1) //pinta Not Found
```

Si queremos cambiar un valor, solo tenemos que asignar un nuevo valor al índice directamente:

```
http404Error.1 = "Página no encontrada"
print(http404Error.1)
```

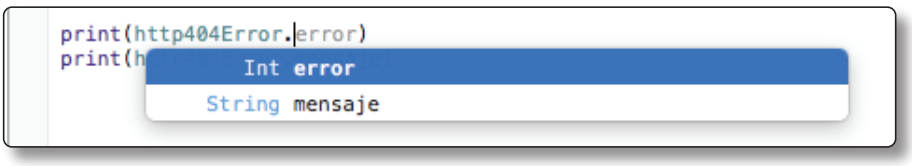
Debemos tener cuidado en no acceder a valores de índice que no existen, ya que provocaría un error.

El uso de índices puede no ser muy fácil de recordar por parte del programador. En caso de tener muchos elementos dentro de la tupla podemos olvidar qué posición ocupa un determinado valor. Para solucionar esto, Swift proporciona la capacidad de nombrar los elementos de la tupla para no tener que recordar el orden en que fueron puestos a la hora de declarar:

```
var http404Error = (error: 404,mensaje: "Not Found")
```

Ahora hemos puesto un nombre a cada uno de los valores de la tupla. Internamente para Swift siguen teniendo un índice por posición (0,1) pero ahora el

programador puede acceder al valor de los elementos introduciendo el nombre; así, el código es más fácil e intuitivo:



1.2.8 Opcionales

Los opcionales se utilizan en situaciones en las que un valor puede estar ausente.

Este concepto no existe en C u Objective-C. Lo más cercano en Objective-C es **nil**, que significa la ausencia de un objeto válido. Sin embargo, esto solo funciona para objetos y no para tipos básicos o valores de enumeración. Para estos tipos, normalmente, devuelven en Objective-C un valor especial (**NSNotFound**) para indicar la ausencia de un valor. Los valores opcionales de Swift permiten indicar la ausencia de un valor de cualquier tipo sin la necesidad de constantes especiales.

Vamos a ver un ejemplo de cómo usar opcionales para hacer frente a la falta de un valor:

```
let numString = "123"
let numInt = Int(numString)
```

Vemos que tenemos una variable **numInt** en la cual estamos intentando transformar un tipo **String** en un **Int.** Esto es un poco peligroso, ya que no todas las cadenas se pueden convertir en un número entero. En este caso se puede, pero con una cadena como, por ejemplo, "Hola mundo" no podríamos.

Debido a que la conversión puede fallar, la función **Int(numString)** devuelve un **Int** opcional, en lugar de un **Int.**, lo que significa que puede contener un valor **int** o ninguno. Se evitan así posibles errores:

```
var variableOpcional : Int?

variableOpcional = 123
variableOpcional = nil
```

Para declarar una variable que pueda contener un valor opcional se pone el símbolo de cierre de interrogación (**?**) después del tipo de variable. En el ejemplo anterior no tenemos un tipo **Int**, sino **Int?**

Podemos observar que la variable opcional puede contener el valor entero o un valor **nil** (un valor vacío). En caso de declarar un variable opcional y no asignar un valor inicial, Swift asigna el valor **nil** de forma automática.

Ahora vamos a pintar el valor de una variable opcional:

```
print(variableOpcional)                    "Optional(123)\n"
```

Podemos observar que al pintar una variable opcional no se pinta el valor, sino que se introduce la palabra **Optional(123)** con el valor en el interior de los paréntesis. Para poder acceder al valor real de una variable opcional tenemos que utilizar el símbolo de cierre de exclamación (**!**) al final de la variable. De esta forma forzamos el desempaquetar el valor de una opcional:

```
if variableOpcional != nil {
    print(variableOpcional!)                "123\n"
}
```

Ahora sí que vemos el valor real de la variable.

El forzar una variable opcional para obtener un valor que contenga **nil** puede provocar un error en tiempo de ejecución, por eso se recomienda comprobar que el valor es distinto a vacío antes de forzar el obtener el valor.

Llegados a este punto ya tenemos los conceptos básicos en el lenguaje Swift:

1. Comentarios
2. Variables y constantes
3. Tipo de variables y declaración
4. Función **print**
5. Números enteros
6. Números de punto flotante
7. **Booleans**
8. Tuplas
9. Opcionales

El siguiente paso son los operadores y expresiones. ¡Tómate un descanso antes de seguir!

1.3 OPERADORES Y EXPRESIONES

Los operadores en Swift son los mismos que en cualquier otro lenguaje, el símbolo que se utiliza para comprobar, cambiar o combinar valores.

Existen tres tipos diferentes de operadores:

1. **Unarios**: aquellos operadores que operan en un solo valor. Los unarios prefijo aparecen inmediatamente antes de su destino (por ejemplo, **!a**); luego tenemos los unarios postfijos sencillos, aquellos que aparecen inmediatamente después de su objetivo (por ejemplo el incremento en uno, **a++**).

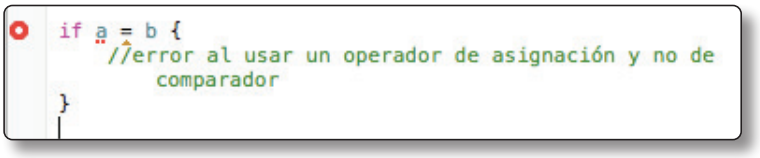
2. **Binarios**: los operadores más comunes, ya que operan con dos valores, por ejemplo, **a + b**.

3. **Ternarios**: operadores que operan en tres valores. En Swift existe un único operador ternario, el operador condicional **a ? b: c**

1.3.1 Operador de asignación

El operador de asignación es aquel que inicializa o actualiza un valor **b** con el de **a (b = a)**.

A diferencia del operador de asignación de C y Objective-C, en Swift no devuelve ningún valor, de esta forma no lo confundimos por error con el operador de comparación (==):



1.3.2 Operadores aritméticos

Los operadores aritméticos son los más comunes y fáciles de usar.

1. Sumar (+)
2. Restar (-)
3. Multiplicar (*)
4. Dividir (/)
5. Resto (**%**)

```
1 + 2 //suma                                    3
6 - 4 //resta                                   2
2 * 2 //multiplicar                             4
20 / 10 //dividir                               2
4 % 2 //el resto de una división                0
```

1.3.3 Operadores de incremento y decremento

Los operadores de incremento y decremento son operadores unarios que aumentan (++) o disminuyen (--) el valor de una variable numérica en uno:

```
var i = 0                                       0
let b = ++i                                     1
let c = i++                                     1
print(i)                                        "2\n"
```

Cada vez que se llama al operador, el valor de la variable se incrementa o disminuye en uno. Los operadores de incremento y decremento son atajos para decir **i = i +1** y **i = i-1**, respectivamente.

Debemos tener en cuenta que estos operadores modifican el valor de la variable en la cual operan y podemos colocar el operador delante o detrás de la variable:

1. **Antes de la variable**: se incrementa la variable antes de devolver su valor.

2. **Después de la variable**: se incrementa la variable después de obtener su valor original.

1.3.4 Operadores de asignación compuestos

Los operadores de asignación compuestos son aquellos que combinan asignación (=) con otra operación:

```
var a = 1                    1
a += 2                       3
```

La operación **a + = 2** es una abreviatura de **a = a + 2**. Lo único que debemos tener en cuenta es que las operaciones de asignación compuestas no devuelven un valor, por lo que no podemos asignar a una variable la operación compuesta **b = a+=2**.

1.3.5 Operadores de comparación

Los operadores de comparación son aquellos que comparan dos valores y devuelven un valor booleano en caso de que se cumpla:

```
2 == 2                       true
2 != 1                       true
5 > 4                        true
2 < 7                        true
1 >= 1                       true
2<=1                         false
```

Los operadores por sí solos no tienen mucho sentido, por lo que se suelen usar sentencias de condición:

```
let edad = 20

if edad >= 18 {
    print("mayora de edad")
}else{
    print("menor de edad")
}
```

```
20


"mayora de edad\n"
```

1.3.6 Operadores lógicos

Los operadores lógicos son aquellos que necesitan de la combinación de dos valores lógicos booleanos **true** y **false**, y devuelven un valor booleano resultante de la operación. En Swift existen los tres operadores lógicos estándar que se encuentran en lenguajes basados en C:

1. **NOT** lógico (**! a**)
2. **AND** lógico (**a && b**)
3. **OR** lógico (**a || b**)

Al igual que ocurre con los operadores de comparación, los operadores lógicos se suelen usar bajo una sentencia que contenga un condicional **if**.

El operador lógico **NOT** invierte un valor booleano donde un valor verdadero se convierte en falso, y a la inversa:

```
let isPlaying = true
if !isPlaying {
    print("PLAY")
}else{
    print("STOP")
}
```

```
true


"STOP\n"
```

El operador lógico **AND** genera un verdadero cuando ambos valores son verdaderos en la expresión. En otros casos siempre es falso:

```
let enteredDoorCode = true                                    true
let passedRetinaScan = false                                  false
if enteredDoorCode && passedRetinaScan {
    print("Welcome!")
} else {
    print("ACCESS DENIED")                                    "ACCESS DENIED\n"
}
```

El operador lógico **OR** se utiliza para crear expresiones lógicas en las que solo uno de los valores tiene que ser verdadero para que genere un valor verdadero. En caso de ser los dos falsos devolverá un valor falso:

```
let hasDoorKey = false                                        false
let knowsOverridePassword = true                              true
if hasDoorKey || knowsOverridePassword {
    print("Welcome!")                                         "Welcome!\n"
} else {
    print("ACCESS DENIED")
}
```

Se pueden combinar múltiples operadores lógicos para crear expresiones compuestas más largas:

```
if enteredDoorCode && passedRetinaScan || hasDoorKey ||
    knowsOverridePassword {
    print("Welcome!")                                         "Welcome!\n"
} else {
    print("ACCESS DENIED")
}
```

En resumen, los operadores en Swift son los siguientes:

1. De asignación (=)
2. De comparación (==, >, <, >=, <=, **!**=)
3. Aritméticos (+, -, *, /, **%**)
4. De asignación compuestos (+=, -=, *=, /=)
5. De incremente y decremento (++, —)
6. Lógicos (**&&**, ||)

1.4 TIPOS DE COLECCIONES

En Swift tenemos los tres tipos de colecciones básicas: matrices, conjuntos y diccionarios para almacenar colecciones de datos.

- **Arrays**: son matrices ordenadas por un índice de valores:

```
var alumnos:[String] = ["Alumno 1","Alumno 2","Alumno 3"]
//array con tres valores  0           1           2
print(alumnos[0])
//pintamos el valor de la posición 0
```

- **Set**: conjuntos de colecciones no ordenados de valores únicos:

```
var escuelas: Set<String> = ["Escuela A", "Escuela B", "Es-
cuela C"]
//conjunto con tres elementos
if(escuelas.contains("Escuela A")){
    print("Tenemos la Escuela A")
}
```

- **Dictionary**: diccionarios de colecciones no ordenados con asociaciones clave-valor:

```
var jugadores: [Int:String] = [1:"Portero",2:"Defensa"]
//los diccionarios tenemos una clave única y su valor
print(jugadores[1]!)
```

El uso de cada uno de ellos depende de la naturaleza de los conjuntos de datos. Si la ordenación es importante y los datos se pueden repetir, se suele usar una matriz. En los casos en que sabemos de antemano que los datos son únicos, usamos el conjunto; y en aquellos casos en que los valores van asociados por una clave única, usamos el diccionario.

Ahora vamos a estudiar cada uno de ellos y algunas de sus características básicas.

1.4.1 Arrays

Un *array*, o matriz, almacena valores del mismo tipo en una lista ordenada. El mismo valor puede aparecer en una matriz en diferentes posiciones.

Podemos crear una matriz vacía indicando el tipo de valores que puede almacenar:

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

De esta forma hemos creado una matriz que puede almacenar valores enteros. Si quisiéramos una matriz que pudiera almacenar cualquier otro tipo de datos, tendríamos que indicarlo entre corchetes ([ ]; por ejemplo, [Tipo]). De momento tenemos una matriz vacía. En ocasiones nos podemos encontrar en alguna situación en la que solo queramos definir la matriz y más adelante introducir valores:

```
matriz.append(23)
```

Con la función **append(valor)** podemos añadir elementos a la matriz. Ahora tenemos una matriz que contiene una posición 0 con el valor 23 y ya podemos pintar su contenido indicando la posición del valor que queremos obtener:

```
print(matriz[0])
```

Observamos que el acceso y modificación de una matriz es un proceso simple. Vamos a ver otras características interesantes.

En muchas ocasiones nos interesa conocer el número de elementos que contiene la matriz para, de esta forma, saber si contiene datos o no. Para esto tenemos dos propiedades muy útiles:

1. **count**: propiedad para obtener el número de elementos de la matriz.

2. **isEmpty**: puede valer **true** o **false** para indicar si la matriz contiene o no datos.

```
var listaCompra = [String]()

listaCompra.append("Tomates")
listaCompra.append("Pizza")

print(listaCompra.count) //al contar tenemos dos elementos

if listaCompra.isEmpty {
    print("Lista vacía")
}else{
    print("Tenemos \(listaCompra.count) elementos")
}
```

Ya conocemos **append()** para añadir un elemento al final de la matriz, pero también podemos utilizar una operación de asignación compuesta para añadir muchos elementos en una misma sentencia (+=).

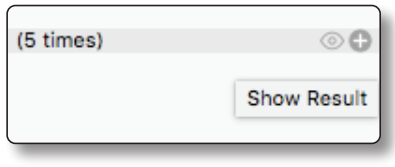
```
listaCompra += ["Queso","Tostadas","Chocolate"]
print(listaCompra.count) //Tenemos 5 elementos
```

Para modificar un elemento de la matriz tenemos que indicar la posición y el nuevo valor:

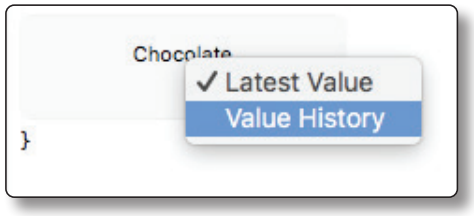
```
print(listaCompra[0]) //pinta tomates
listaCompra[0] = "Agua" //modificamos valor
print(listaCompra[0]) //pinta agua
```

Finalmente, para recorrer todos los elementos de una matriz utilizamos un bucle **for** muy sencillo:

Para poder mostrar el resultado de un bucle al recorrer una matriz tenemos que presionar sobre el símbolo + que aparece al lado del número de iteraciones, a la derecha del Playground:



De esta forma nos aparecerá el último resultado del recorrido. Finalmente, presionando el botón derecho sobre el resultado podemos seleccionar **Ver todos los datos**:



1.4.2 SET

Un conjunto almacena valores distintos de un mismo tipo en una colección sin orden definido. Se puede utilizar un conjunto en lugar de una matriz cuando el orden de los elementos no es importante o cuando es necesario asegurarse de que un elemento solo aparece una vez.

Al igual que ocurre con las matrices, podemos definir un conjunto vacío para un tipo de dato determinado:

```
var letras = Set<Character>()
```

En los conjuntos tenemos que indicar el tipo de dato de la colección entre antilambdas (son los símbolos < y >; por ejemplo, <tipo>). Ahora, para insertar un nuevo elemento al conjunto utilizamos la función **insert(valor)**:

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

Podemos acceder y modificar un conjunto a través de sus métodos y propiedades.

Utilizamos la propiedad **count** para averiguar el número de elementos del conjunto; y con **isEmpty** sabemos si es un conjunto vacío o no:

```
if letras.isEmpty {
    print("Conjunto vacío")
}else {
    print("Conjunto de \(letras.count) elementos")
}
```

Para eliminar un elemento del conjunto utilizamos el método **remove(valor)**, pero primero es recomendable saber si contiene un valor concreto con la función **contains(valor)**:

```
if letras.contains("a"){
    letras.remove("a")
}else {
    print("No tenemos la letra a")
}
```

Finalmente, para recorrer todos los elementos de un conjunto se utiliza el mismo bucle que en las matrices [antes podemos ordenar la lista con **sort()**]:

```
letras.insert("b")

for letra in letras.sort(){
    print(letra)
}
```

1.4.3 Diccionarios

Un diccionario almacena un conjunto de valor por asociación clave-valor. Utiliza un tipo de dato para especificar la clave, siendo valores únicos, y cada clave tiene vinculado un valor con un tipo de dato. A diferencia de las matrices, los elementos de un diccionario no tienen un orden específico. Se utiliza un diccionario cuando se necesita buscar valores en función de su identificación.

Para crear un diccionario vacío tenemos que indicar el tipo de dato para la clave y el tipo de dato para el valor:

```
//Dicionario
var provincias = [String:String]()
```

Podemos crear un diccionario con valores iniciales: **[clave1: valor1, clave2: valor2…]**:

```
// nombre :  [Clave : Valor] = ["CLAVE":"VALOR...]
var paises : [String:String] = ["ES":"España","FR":"Francia",
"IT":"Italia"]
```

Al igual que ocurre con las matrices y los conjuntos, en los diccionarios tenemos disponibles dos propiedades para saber el número de elementos y si contiene datos o no, **count** y **isEmpty**:

```
//count
print("Tenemos \(paises.count) países")

//isEmpty
if paises.isEmpty{
    print("No tenemos países registrados")
}else{
    print("Existen países")
}
```

Añadir un nuevo elemento es tan sencillo como poner entre corchetes ["clave"] y una operación de asignación:

```
paises["GER"] = "Alemania"
```

Si la clave insertada es nueva, creará un nuevo elemento; si ya existe, la editará. En el caso de querer pintar un valor tenemos que indicar la clave del valor correspondiente:

```
print(paises["ES"])
```

Para eliminar un elemento del diccionario tenemos dos opciones, o igualamos a **nil** el valor de una clave específica o utilizamos la función **removeValueForKey()**:

```
//igualamos españa a nil
paises["ES"] = nil
//eliminamos italia
paises.removeValueForKey("IT")
//eliminamos todos
paises.removeAll();
```

Para poder recorrer todos los datos de un diccionario utilizamos el mismo mecanismo ya estudiado en las matrices y conjuntos. El bucle **for** nos permite recorrer uno a uno cada elemento del diccionario:

```
for (paisCode,paisName) in paises {
    print("\(paisCode) : \(paisName)")
}
```

En un diccionario tenemos la opción de recorrer todas las claves que contiene o sus valores de forma separada:

```
//Pintamos claves
for paisCode in paises.keys{
    print(paisCode)
}
```

```
//Pintamos valores
for paisName in paises.values{
    print(paisName)
}
```

Ya hemos estudiado de forma superficial los tres principales tipos de colecciones: *arrays*, **Set** y **Dictionary**.

Recordemos: son elementos capaces de almacenar un conjunto de valores, cada uno con unas características distintas:

1. **Propiedades**: con la propiedad **count** sabemos el número de elementos. Con **isEmpty** obtenemos un booleando indicando si está vacío o no.

2. **Arrays**: matriz con un conjunto de valores ordenado y por índices. Los valores se pueden repetir.

3. **Set**: conjunto de valores desordenados y únicos.

4. **Diccionario**: asociación clave-valor, sin un orden específico. Las claves son únicas, pero no sus valores, ya que se pueden repetir en dos claves distintas.

1.5 CONTROLES DE FLUJOS

Llegados a este punto tenemos claras tres cosas en Swift:

1. Variables y constantes

2. Operadores

3. Colecciones

Y ahora vamos a estudiar los controles de flujo, la última pieza para completar el lenguaje lineal de Swift.

En los controles de flujo tenemos dos grandes divisiones:

1. Bucles **for** y **while**

2. Condicional **if** y **swich**

Los bucles **for** ya los hemos visto para recorrer colecciones, pero ahora vamos a profundizar un poco más. También conoceremos y aprenderemos el uso del bucle **while** y sus variantes. Finalmente, terminaremos por conocer al famoso **if** y el control de flujo por casos, el **swich**.

1.5.1 Bucle for

Swift ofrece dos tipos de bucles para llevar a cabo un conjunto de instrucciones un cierto número de veces:

1. El **for – in** realiza un conjunto de sentencias para cada elemento de una secuencia:

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

2. El **for** realiza una serie de declaraciones hasta que se cumpla una condición específica; por lo general, mediante el incremento de un contador:

```
for var i = 0; i < 20; i++ {
    print(i)
}
```

Usaremos el **for - in** en aquellos casos en los que queramos repetir una secuencia, tales como rangos de números, elementos de una matriz o caracteres de una cadena.

En el **for - in** tenemos que indicar un nombre para el índice y un número de iteraciones. Para el número de iteraciones tenemos dos opciones:

```
for i in 1 ... 5 {
    print("\(i) por 5 es: \(i*5)")
}
```

La primera es indicar el número inicial seguido de tres puntos y el número final. La otra opción es indicar la colección que queramos recorrer, por lo que el número de iteraciones dependerá del número de elementos de la colección:

```
let animales = ["Perro","Gato","Serpiente"]

for animal in animales {
    print(animal)
}
```

En Swift existe el bucle **for** tradicional, con una condición y un incremento. El formato para un **for** es:

```
for  inicialización ; condición ; incremento {
   sentencias
}
for var i = 0; i < 3; i++ {
    print(i)
}
```

Los bucles funcionan de la siguiente forma:

1. La primera vez inicializa la variable.

2. Comprueba que cumple la condición. En caso de que la cumpla, realiza el paso 3; en caso de que no la cumpla, finaliza el bucle.

3. Realiza las sentencias.

4. Opera sobre la variable inicializada.

5. Volvemos al paso 2.

1.5.2 Bucle while

Un bucle **while** realiza un conjunto de declaraciones hasta que una condición se convierte en **false**. A diferencia de los bucles **for**, los bucles **while** se suelen usar cuando no se conoce el número de iteraciones que se han de realizar.

En Swift existen los dos tipos bucles **while**:

1. **While**: primero comprueba una condición y si se cumple realiza las sentencias hasta que ya no cumpla.

```
while (i<20){
    print("hola")
    i++
}
```

2. **repeat – while**: primero realiza las sentencias y luego comprueba la condición.

```
var x = 0;

repeat {
    print("hola")
    x++
}while (x<20)
```

Es decir, el **while** pregunta y realiza; el **repeat while** realiza y luego pregunta.

El **while** se inicia mediante la evaluación de una sola condición. Si la condición es verdadera, se realiza un conjunto de sentencias y se repiten hasta que la condición se convierte en falsa.

El **repeat – while** realiza una sola pasada a través del bloque la primera vez, antes de considerar la condición del bucle. A continuación, sigue repitiendo el bucle mientras la condición sea verdadera.

1.5.3 Condicionales

La programación es una constante toma de decisiones; de manera que si se cumple una condición, se ejecuta una acción u otra.

Vamos a poner un pequeño ejemplo que podemos encontrar en la vida cotidiana. Imaginemos que queremos comprar unos pantalones en una tienda. Realizamos las sentencias de acceder a la tienda, vamos a la sección de pantalones y empezamos a mirar la lista. Cogemos un pantalón y realizamos una condición. ¿Nos gusta el pantalón? En caso afirmativo, lo cogeremos de la percha; en caso negativo, lo dejaremos estar y seguiremos buscando (bucle). En caso de que nos guste tendremos el pantalón en nuestra mano y miraremos la talla. ¿La talla es la correcta? En caso afirmativo, seguiremos con el pantalón; en caso negativo, lo dejaremos y volveremos a realizar las acciones anteriores. Finalmente, siguiendo este patrón de condiciones podemos llegar a salir de la tienda con un pantalón en nuestra bolsa, o no.

Todo esto se ha hecho mediante condicionales. En la programación el proceso es exactamente el mismo. En el caso de Swift tenemos los condicionales **if** y **Swich**.

El **if**, en su forma más simple, tiene un solo caso de condición. Se ejecuta un conjunto de instrucciones solo si la condición se cumple:

```
let num:Int = 2;
if num == 2 {
    print("Cumple la condición")
}
```

Podemos observar cómo analiza una condición utilizando una operación de comparación. En caso de cumplirse dicha condición, el **if** realizará el conjunto de sentencias que se encuentran entre sus llaves.

Si queremos realizar alguna acción en caso de que no se cumpla la condición, utilizaremos la sentencia **else**:

```
var edad :Int =  11
if edad >= 18 {
    print("Mayor de edad")
}else{
    print("No puedes acceder")
}
```

En este caso mostrará un mensaje de bienvenida en caso de ser mayor de edad, en caso contrario mostrará un mensaje indicando que no puedes acceder.

Si queremos analizar más de una condición, utilizamos el condicional **else if**. Swift va analizando todas las condiciones que encuentra en el **if**; en caso de no cumplir la primera, pasa a la segunda, y así sucesivamente. En caso de encontrar una condición verdadera accede a las llaves, realiza las sentencias de código y finaliza el condicional **if**, sin llegar a analizar el resto de condiciones. Podemos terminar el condicional con la sentencia **else** en caso de que no cumpla con ninguna condición (prueba a modificar el valor de la variable):

```
var pais:String = "ES"
if pais == "ES"{
    print("Nacinalidad española")
}else if pais == "IT"{
    print("Nacionalidad italiana")
}else if pais == "FR"{
    print("Nacionalidad francesa")
} else
    print("No conocemos la nacionalidad")
}
```

En muchas ocasiones no utilizamos el **else if** ya que parece que conseguimos lo mismo utilizando el **if** suelto:

```
var pais:String = "ES"
if pais == "ES"{
    print("Nacinalidad española")
}
if pais == "IT"{
    print("Nacionalidad italiana")
```

```
}
if pais == "FR"{
    print("Nacionalidad francesa")
}
```

Pero en lo que se refiere al rendimiento no es lo mismo, ya que en el caso de utilizar **else if**, cuando una condición se cumple deja de analizar el resto de condicionales, y de esta forma el rendimiento mejora notablemente. Si tenemos las condiciones en **if** sueltos, se analizará cada uno de ellos aun sabiendo que no se van a cumplir.

Cuando vemos que, por la naturaleza del programa que estamos creando, utilizamos muchos condicionales con **else if** podemos plantearnos utilizar un **Swich**.

Un **Swich** considera un valor y lo compara con varios patrones coincidentes posibles. A continuación, ejecuta un bloque apropiado de código, basado en el primer patrón que coincide con éxito.

En su forma más simple, un **Swich** compara un valor con uno o más valores del mismo tipo:

```
switch some value to consider {
case value 1:
    respond to value 1
case value 2,
value 3:
    respond to value 2 or 3
default:
    otherwise, do something else
}
```

El **swich** consiste en múltiples posibles casos, cada uno de los cuales comienza con la palabra reservada **case**.

Debemos tener en cuenta que un **swich** analiza cada uno de los casos hasta encontrar una coincidencia de valor, realizando el bloque de sentencias encontrado

en él. A continuación deja de analizar los casos y realiza bloque por bloque cada uno de los casos encontrados a continuación, a no ser que se encuentre dentro del caso con una sentencia **break**, la cual rompe el **swich** y termina el bloque. El **default** es aquel que ejecutará en caso de no encontrar ninguna coincidencia en el resto de casos, es comparable con el **else**:

```
var pais:String = "ES"
switch pais {
case "FR" : print("Nacionalidad francesa")
            break
case "ES" : print("Nacionalidad española")
            break
case "IT" : print("Nacionalidad italiana")
            break
default: print("No conocemos la nacionalidad")
}
```

Los condicionales se pueden complicar mucho más, pero no es el objetivo del libro profundizar en el lenguaje, por lo que se recomienda, en caso de curiosidad, investigar la documentación oficial de Apple y realizar pruebas con distintos casos. Al final de este tema encontrarás un conjunto de ejercicios y retos para poder aumentar tus conocimientos sobre el lenguaje Swift.

Recordemos que la mejor forma de aprender un lenguaje es realizando proyectos reales, prácticas y pequeños experimentos con él. Los años de experiencia y la cantidad de programas creados condicionan los conocimientos que podemos llegar a tener sobre un lenguaje.

Hemos aprendido a utilizar las herramientas básicas para poder programar pequeños programas lineales. Es decir, programas que empiezan por la primera línea y terminan en la última. Pero en el mundo de iOS esta forma de programar no es la real, ya que existen eventos, saltos en las líneas de código, acceso a otros ficheros y fragmentos, etc. Por esto, en los siguientes apartados del tema aprenderemos a crear programas modulares, programas que no siguen un patrón lineal.

1.6 FUNCIONES

Las funciones son bloques de código que ejecutan una acción específica. Elementos autónomos con un nombre que los identifique para poder ser llamados y realizar su tarea cuando sea necesario.

Las funciones pueden tener o no parámetros de entrada, variables necesarias para el correcto funcionamiento de la función, y pueden (o no) devolver un valor resultante al ejecutar la función.

Cada función tiene un nombre único que la identifica, y se define utilizando la palabra reservada **func**:

```
/*
FUNCIONES
*/
//func nombreFunción ()
func decirHola () {
    //SENTENCIAS DE CÓDIGO QUE REALIZAN UNA ACCIÓN
    print("Hola Swift")
}
//llamar a la función por su nombre
decirHola()
```

Ahora podemos definir una función que pida un parámetro de entrada. En la variable de entrada indicamos un nombre y un tipo de valor:

```
//func nombreFunción (parámetro de entrada)
func decirHola (persona: String) {
    //SENTENCIAS DE CÓDIGO QUE REALIZAN UNA ACCIÓN
    print("Hola \(persona)") //podemos usar el parámetro de
entrada
}

//llamar a la función por su nombre y añadiendo un valor al
parámetro de entrada
decirHola("Kike")
```

Funciones con más de un parámetro de entrada:

```
//func nombreFunción (parámetros de entrada separados por ,)
func decirHola (nombre: String, apellidos: String) {

    print("Hola \(nombre) \(apellidos)") //podemos usar los
parámetros de entrada
}
```

Para llamar a una función tenemos que introducir su identificación seguida de los paréntesis. En los paréntesis introducimos los valores que van a obtener sus variables de entrada en caso de que los pida:

```
//llamar a la función por su nombre y añadiendo un valor a
los parámetros de entrada
decirHola("Kike",apellidos: "Blasco Blanquer")
```

En muchas ocasiones queremos crear funciones que tras cumplir su objetivo nos devuelvan un valor resultante de sus acciones. Para realizar esto utilizamos la palabra reservada **return** y en la función indicamos el tipo de valor de retorno utilizando el símbolo -> Tipo:

```
func sumar (numA : Int, numB : Int) -> Int{

    return numA + numB

}
```

En la línea de código en la que realizamos la llamada obtendremos el valor resultante de la función:

```
var resultado = sumar(5, numB: 6)
```

Algo que se debe destacar de las funciones Swift es el retorno de múltiples valores. Vamos a observar el funcionamiento en estos casos:

```
//En el tipo de retorno podemos poner más de uno entre () se-
parados por ,
func minMax(numeros: [Int]) -> (min: Int,max : Int){

    var min = numeros[0]
    var max = numeros[0]

    for num in numeros{
        if num < min {
            min = num
        } else if num > max{
            max = num
        }
    }

//tenemos que devolver tantos valores como elementos de sali-
da tengamos
    return (min,max)
}
//Array de números
let numeros:[Int] = [18,6,2004]
//Almacenamos los valores de retorno al llamar a la función
var valores = minMax(numeros)
//Con punto indicamos la variable que queramos obtener
print(valores.min)
print(valores.max)
```

1.6.1 Nombre de parámetro externo y local

Los parámetros de función tienen tanto un nombre de parámetro externo como un nombre de parámetro local. El nombre de parámetro externo se utiliza para etiquetar los argumentos pasados a una llamada de función. El nombre de parámetro local se utiliza en la aplicación de la función:

```
func nombreFuncion(primerParam : Int, segundoParam : Int){
    //El cuerpo de la función:

    //primerParam y seguñndoParam son los nombres
    //locales para hacer referencia a las variables
    //dentro de la función

}
```

Por defecto, el primer parámetro omite su nombre externo. Los siguientes parámetros usan su nombre local como su nombre externo a la hora de ser llamado:

```
nombreFuncion(1, segundoParam: 2)
```

Podemos especificar un nombre de parámetro externo introduciendo primero el nombre externo y luego el nombre local de la variable separado por un espacio:

```
func nombreFuncion(nombreExterno nombreLocal : Int){
    //El cuerpo de la función:

    //nombreLocal es el que usamos en la función

}
//nombreExterno el que usamos fuera de la función
nombreFuncion(nombreExterno: 15)
```

Otro ejemplo:

```
func decirHola (to persona : String, and otraPersona :
String){
    print("Hola \(persona) y \(otraPersona)")
}
decirHola(to: "Mar", and: "David")
```

Podemos omitir el nombre externo en el resto de parámetros (siempre que no sean el primero) introduciendo un guion bajo (_) delante del nombre local:

```
func decirHola (persona : String, _ otraPersona : String){
    print("Hola \(persona) y \(otraPersona)")
}
decirHola("Mar","David")
```

Por último, podemos crear funciones con parámetros de entrada por defecto:

```
func nombreFuncion( valorPorDefecto : Int = 4){
    //El cuerpo de la función:
    print(valorPorDefecto)

}
nombreFuncion() //No hace falta indicar un valor
nombreFuncion(6) //Podemos añadir un valor
```

Ahora ya podemos realizar nuestros programas un poco más estructurados. Utilizando funciones podemos reutilizar el código y hacer que quede mucho más limpio. También conseguimos que nuestros programas no sean lineales, pues será posible saltar entre bloques de código según se realicen las llamadas.

1.7 PROGRAMACIÓN ORIENTADA A OBJETOS

Para entrar en el mundo de la programación en iOS es importante que conozcamos un poco, ya que es un mundo muy extenso, la sintaxis en la programación orientada a objetos, es decir, saber cómo realizar clases, la herencia, instanciar un objeto, los constructores, etc. En un programa real de iOS nos encontraremos con este paradigma de programación usando objetos en sus interacciones.

Antes de abordar este estudio, es necesario que aclaremos una serie de conceptos importantes sobre clases y objetos, que no solo nos facilitarán la comprensión del lenguaje Swift, sino que nos permitirán también ir estructurando poco a poco nuestros programas de una forma más adecuada. Posteriormente, analizaremos en profundidad los conceptos de la programación orientada a objetos y su aplicación en el lenguaje Swift.

1.7.1 Clases

Como ya hemos mencionado, todo programa en iOS se estructura en clases; pero ¿qué es exactamente una clase?

Desde un punto de vista conceptual, una clase define el comportamiento de un determinado tipo de elemento y se trata simplemente de eso, de un conjunto de especificaciones que determinan cómo se van a comportar cierto tipo de elementos. Estos elementos, creados a partir de las especificaciones de la clase, son lo que llamamos *objetos*.

En resumen, la clase define el comportamiento y características de cierto tipo de elementos, mientras que los elementos físicos, creados a partir de las especificaciones de la clase, serían los objetos.

En programación las clases contienen el código que define el comportamiento de los tipos de objeto que representan. Este código está dividido en dos partes:

1.7.2 Atributos

Representan las características que tendrán los objetos de la clase, como el color, precio, potencia, etc.

Normalmente, los atributos son manejados a través de variables internas de la clase.

1.7.3 Métodos

Se trata de funciones que implementan la funcionalidad de los objetos de la clase. En el caso de una hipotética clase **coche**, algunos de los métodos que determinan el funcionamiento de los mismos serían acelerar o frenar.

En Swift, las clases se implementan mediante la palabra reservada **class**, incluyendo en su interior un bloque de sentencias con los atributos y métodos que la definen. La estructura típica de una clase es la siguiente:

```
class NombreClase {
    //Propiedades:

    //Métodos:

}
```

Los atributos de una clase Swift se definen a través de variables de ámbito privado, es decir, variables que únicamente pueden ser utilizadas desde el interior de la clase. Por su parte, los métodos estarán implementados mediante funciones, las cuales podrán devolver o no un valor, en función de la naturaleza del método, y podrán recibir parámetros.

El formato de un método o función en Swift ya lo conocemos.

A fin de que, una vez creado el objeto de la clase, sus métodos puedan ser utilizados desde cualquier parte del código, se debe utilizar el modificador de acceso **public** en la declaración de los mismos.

Como hemos mencionado, se recomienda declarar a los atributos de una clase como **private**, a fin de que el acceso a ellos solo pueda realizarse desde el interior de la clase (a este concepto se le conoce como *encapsulación*).

Si queremos proporcionar acceso a los atributos desde el exterior de la clase, habrá que hacerlo a través de métodos públicos que proporcionen un acceso controlado a los mismos. Por cada uno de los atributos habrá que proporcionar una pareja de métodos, conocidos como métodos **set**/**get**, que nos permitan realizar las operaciones de escritura y lectura, respectivamente, sobre los atributos:

```
class NombreClase {
    //Propiedades:
    private var prop1 : Int?

    //Funciones get and set:
    func getProp1() -> Int{
        return prop1!
    }
    func setProp1(valor:Int){
        prop1 = valor
    }

}
```

Vamos a dedicar un tiempo a observar la clase creada. Primero, hemos declarado todas las propiedades de la clase arriba con variables. Observamos que hemos tenido que poner el interrogante en la declaración del tipo, ya que el no asignar un valor en su asignación significa que puede contener un valor vacío, y, recordemos, en Swift los valores vacíos son los opcionales.

A continuación se han definido dos funciones: una con un valor de retorno, llamada **get**, cuya función es la de devolver el valor de una propiedad privada de la clase. Introducimos la exclamación al final de la variable para obtener el valor de la variable opcional. La segunda función, conocida como **set**, contiene un parámetro de entrada y es la encargada de modificar el valor de una propiedad privada de la clase.

El uso de funciones **get**/**set** no es obligatorio, pero es una buena forma de proteger las propiedades de las clases y poder acceder a ellas desde el exterior de forma controlada. El resto de métodos de la clase se definen a continuación, dotando de funcionalidad a la clase. Estos conceptos los iremos estudiando poco a poco en los siguientes temas mientras vamos aprendiendo a desarrollar para iOS

Una vez implementada la clase, ya está lista para ser utilizada por los programas. Con la inclusión de atributos y métodos dentro de una clase conseguimos que toda esta funcionalidad, definida en un único lugar, pueda reutilizarse tantas veces como sea necesario en cualquier parte del programa.

La utilización de la funcionalidad de una clase requiere la creación de un objeto o instancia de la misma. Este objeto será referenciado a través de una variable con la que podremos llamar a los métodos definidos dentro de la clase:

```
//Instanciar un objeto de la clase.
var objeto : NombreClase = NombreClase();
//Ahora podemos usar sus métodos
objeto.setProp1(2)
print(objeto.getProp1())
```

1.7.4 Inicialización

Normalmente, los objetos de una clase requieren que parte de sus atributos sean inicializados con un valor antes de poder llamar a los métodos del mismo. Por ejemplo, supongamos que tenemos la siguiente versión de la clase **Calculadora** con un método que nos permite calcular potencias de un número elevado a otro dado:

```
public class Calculadora {
    //Propiedades
    private var exponente : Int?;
    //Funciones
    func setExponente(exp : Int){
         exponente=exp;
    }
    func  potencia(base : Int) -> Int{
        var resultado : Int = 1;
        for var i=1;i<=exponente;i++ {
            resultado *= base;
        }
        return resultado;
   }
}
```

Dado que el exponente está definido como un atributo de la clase, será necesario establecer un valor en el mismo antes de llamar al método **potencia()** encargado de realizar el cálculo. Por ejemplo:

```
let calculadora : Calculadora = Calculadora()
calculadora.setExponente(2)
print(calculadora.potencia(3))
```

La incomodidad que supone llamar explícitamente a los métodos de inicialización de atributos puede ser eliminada gracias a la utilización de los inicializadores.

El inicializador **init()** es el proceso de preparación de una instancia de la clase, estructura o enumeración para su uso. Este proceso implica el establecimiento de un valor inicial para cada propiedad almacenada y llevar a cabo cualquier otra configuración o inicialización:

```
public class Calculadora {
    //Propiedades
    private var exponente : Int?;


    //Inicializador
    public init(exp : Int){
        exponente = exp
    }


    //Funciones
    func setExponente(exp : Int){
         exponente=exp;
    }

    func  potencia(base : Int) -> Int{
        var resultado : Int = 1;
        for var i=1;i<=exponente;i++ {
            resultado *= base;
        }
    return resultado;
    }

}
```

En la misma instrucción de creación del objeto, se debe especificar entre paréntesis el valor del argumento o argumentos de llamada al inicializador:

```
//ahora estamos obligados a inicializar el valor exponenete
let calculadora : Calculadora = Calculadora(exp: 2)
print(calculadora.potencia(3))
```

1.7.5 Destructor

Un destructor se llama inmediatamente antes de que se cancele la asignación de una instancia de clase. Para definir un destructor utilizamos la instrucción **deinit**:

```
//Destructor
    deinit{
        //Cuerpo de la función
    }
```

Swift realiza automáticamente una limpieza de memoria para liberar recursos, pero en ocasiones, cuando se trabaja con clases propias, es posible que necesitemos alguna limpieza adicional. Por ejemplo, si creamos una clase personalizada para abrir un archivo y escribir algunos datos en él, es posible que tengamos que cerrar el archivo antes de que se cancele la asignación de la instancia de la clase.

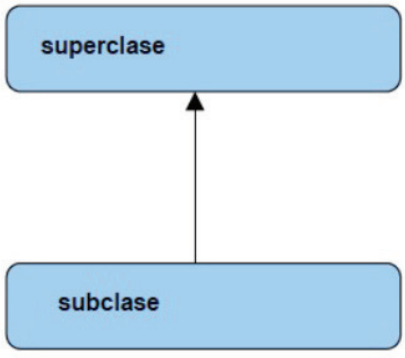
1.7.6 Herencia

La herencia es quizá la característica más interesante y potente que ofrecen los lenguajes orientados a objetos. Mediante ella es posible crear clases que dispongan de forma automática de todos los miembros (atributos y métodos) definidos en clases ya existentes.

Esto es particularmente útil en aquellos contextos en los que necesitamos utilizar una clase con los métodos incluidos en otra ya existente, pero a la que queremos añadir una nueva funcionalidad; en vez de modificar la clase original, emplearemos la herencia para crear una nueva clase con todos los métodos definidos en la primera y sobre ella incluir los nuevos elementos que se necesiten.

Como se desprende de lo comentado anteriormente, la herencia representa un excelente mecanismo de reutilización de código, pues incorpora en las nuevas clases los métodos definidos en otras sin tener que rescribirlos.

La relación de herencia entre dos clases se expresa gráficamente como se indica en la figura, mediante una flecha que sale desde la clase que hereda, conocida también como *subclase*, y que apunta a la clase heredada, llamada también *superclase*.



Cuando creamos una clase Swift y queremos que esta clase herede de otra existente, debemos utilizar dos puntos junto con el nombre de la superclase justo al lado del nombre de la subclase:

```
public class SuperClase {
    //Propiedades
    var prop1 : Int?
    //Métodos
    func metA (){

    }
}


public class SubClase : SuperClase{
    //Propiedades
    var prop2 : Int?
    //Métodos
    func metB(){

    }

}

var objeto : SubClase = SubClase()

objeto.metA() //método heredado
objeto.metB() //método propio
```

Hay que indicar que, a pesar de que la subclase hereda todos los miembros definidos en la superclase, los miembros que estén definidos dentro de esta como **private** no serán accesibles directamente desde la subclase.

1.7.7 Sobrescritura de métodos

En determinados contextos de herencia, no todos los métodos heredados por la subclase se ajustan a los requerimientos de la misma. Puede ocurrir que alguno de los métodos heredados deba ser redefinido en la nueva clase para poder cumplir mejor con su funcionalidad.

A esta redefinición de métodos heredados en la subclase se la conoce como *sobrescritura de métodos* y su objetivo es volver a definir en la subclase un método que ha sido heredado de la superclase respetando el formato original del mismo. Esto

significa que el nuevo método tiene que tener exactamente el mismo nombre, tipo de devolución y lista de parámetros definidos en la superclase.

Para sobrescribir una función de la superclase tenemos que añadir la palabra reservada **override** delante de la función:

```
public class SubClase : SuperClase{
    //Propiedades
    var prop2 : Int?
    //Métodos
    func metB(){

    }

    override func metA() {
        //nueva funcionalidad del método heredado
    }

}
```

Ahora según el objeto que llame la función realizará una funcionalidad u otra:

```
var objeto : SubClase = SubClase()

objeto.metA() //método heredado en la versión sobrescrita


var otroObjeto : SuperClase = SuperClase()

otroObjeto.metA() //método original
```

Podemos llamar a la versión de un método de la superclase dentro de la versión sobrescrita de la subclase utilizando la palabra reservada **super**. Cuando ponemos **super** estamos haciendo referencia a la superclase.

```
override func metA() {
        //Llamada a la función orginal de la superclase
        super.metA()
        //nueva funcionalidad del método heredado
    }
```

1.7.8 Protocolos

Un protocolo es un conjunto de métodos, propiedades y otros requisitos sin definir.

Su objetivo es simplemente definir cómo tiene que ser el formato de determinados métodos y propiedades que deben existir en determinados tipos de clases.

Los protocolos son implementados por una clase, estructura o enumeración para proporcionar una implementación de dichos métodos.

La sintaxis para la definición de un protocolo es la siguiente:

```
//Definición de un protocolo

protocol PrimerProtocol {
    func metodoA()
    func metodoB()
}
```

Para obligar a una clase a codificar los métodos del protocolo respetando el formato establecido, dicha clase deberá extender el protocolo:

```
//Una clase tiene que implementar un protocolo
public class ClaseA : PrimerProtocol {

    //Implementamos los métodos del protocolo
    func metodoA() {
    }

    func metodoB() {
    }
}
```

1.8 EJERCICIOS PROPUESTOS

La programación como tal tiene muchos más conceptos y se necesita un estudio más profundo para llegar a entender correctamente cada una de las herramientas de las que disponemos. Pero con los conceptos teóricos vistos hasta el momento tenemos suficiente para empezar a entender el desarrollo de aplicaciones con iOS. A lo largo del libro entenderás mejor los conceptos vistos hasta el momento, y, sobre todo, verás una utilidad real de cada uno de ellos. Se recomienda volver

a revisar este tema tantas veces como sea necesario si algún concepto teórico de programación no se ha entendido bien.

La mejor forma para aprender a programar y a crear aplicaciones es realizar muchas prácticas, ejercicios, proyectos, etc. Y se necesita, como para todo en la vida, de un tiempo de aprendizaje y adaptación. Tener paciencia, confianza y saber ser comprensivo con uno mismo son las herramientas, a mi parecer, que nos ayudarán a alcanzar nuestras metas.

Ejercicio 1

Realizar una función en la que se le pasen dos números y los pinte ordenados de mayor a menor.

Ejercicio 2

Función en la que se le pasen tres números y los pinte ordenados de menor a mayor.

Ejercicio 3

Función en la que se introduce una nota y devuelve el resultado en forma de INSUFICIENTE, SUFICIENTE, BIEN, NOTABLE, EXCELENTE.

Ejercicio 4

Realizar una función que necesite el día, mes y año de una fecha e indique si la fecha es correcta. Con meses de 28, 30 y 31 días. Sin años bisiestos.

Ejercicio 5

Función a la que se le pase un número *N* y pinte por pantalla todos los números entre 1 y *N*.

Ejercicio 6

Crear una clase llamada **Calculadora** que contenga cinco métodos: sumar, restar, dividir y multiplicar dos números cualquiera

Ejercicio 7

Crear una clase que herede de **Calculadora** y contenga un método más llamado **resto**, el cual devuelva el resto de una división.

Los conceptos aprendidos hasta el momento son los siguientes:

1. Variables y constantes
2. Operadores
3. Colecciones
4. Bucles
5. Condicionales
6. Funciones
7. Clases
8. Métodos y atributos
9. Encapsulación
10. Instanciar
11. Herencia
12. Métodos sobrescritos
13. Super
14. Protocolos

¡Ahora ya estamos preparados para adentrarnos en el mundo de iOS!