

Swift en Ejemplos

Poste Studio

Published
with GitBook



Tabla de contenido

Introducción	0
Cómo usar este libro	1
Primeros Pasos	2
Introducción al lenguaje	3
Tipos de Datos	3.1
Int	3.1.1
Double	3.1.2
Float	3.1.3
String	3.1.4
Bool	3.1.5
Variables	3.2
Constantes	3.3
Arrays	3.4
Diccionarios	3.5
Funciones	3.6
Tuplas	3.7
Swift Intermedio	4
Control de flujo	4.1
if/else	4.1.1
switch	4.1.2
guard	4.1.3
where	4.1.4
Enumeraciones	4.2
Opcionales	4.3
Ciclos	4.4
Búsqueda de Patrones	4.5
Estructuras	4.6
Clases	4.7
Propiedades	4.8
Tipos de datos de Valor	4.9

Tipos de datos de Referencia	4.10
Extensiones	4.11
Protocolos	4.12
Implementación por defecto de Protocolos	4.13

Bienvenido a Swift en Ejemplos

Swift en Ejemplos tiene un único objetivo: ayudarte a aprender Swift, el nuevo lenguaje de programación de Apple.

A través de ejemplos concisos, aprenderás paso a paso desde temas básicos (como tipos de datos) hasta avanzados (como mejores prácticas).

Antes de comenzar

No importa si no sabes programar en algún lenguaje. Swift es tan sencillo que bien podría ser el lenguaje con el que por fin aprendas a programar. Sin embargo, un conocimiento superficial de conceptos sobre programación es sugerido. Digamos que si tienes una idea de por qué estas leyendo este libro, ya vamos por buen camino.

Puedes encontrar más información sobre el libro y cómo usarlo en [Cómo usar este libro](#).

Acerca De SwiftEnEjemplos.com

[SwiftEnEjemplos.com](#) es mantenido por mi, [Oscar Swanros](#), pero es un trabajo de comunidad.

Como todo en el mundo de la programación, Swift es un lenguaje que está cambiando día con día (más ahora que es un [proyecto de código libre](#)). Este proyecto, de igual forma, es un trabajo en proceso y cualquier aportación es agradecida.

Si SwiftEnEjemplos.com te ha sido de ayuda alguna, te pido por favor que consideres convertirte en un *patrón* y apoyes desde \$1 USD al mes a que SwiftEnEjemplos.com se mantenga siempre vigente. Conoce más [aquí](#).

Nota de Oscar:

Espero que disfrutes SwiftEnEjemplos.com. Pero aún más, que te sea útil y te diviertas leyéndolo tanto como yo divertí escribiendo cada uno de los artículos que lo comprenden.

— O.

- [@SwiftEnEjemplo en Twitter](#)
- <http://swanros.com>

- oscar@swanros.com
- [@Swanros en Twitter](#)

Cómo usar este libro

Los contenidos de este libro están disponibles completamente gratis en su [repositorio de GitHub](#), desde donde puedes descargar todos los contenidos en formato `.zip`.

Versión de Swift

Todos los ejemplos de este libro están probados con la última versión de Swift disponible a través de los canales finales de Apple. Actualmente: **Swift 2.1**.

Errores en los artículos y contribuciones

Todos los artículos en este libro están escritos usando la sintaxis Markdown. Si encuentras algún error, o quieres aportar un artículo en específico, siempre puedes [abrir un Pull Request al repositorio](#). Tu contribución será (posiblemente) editada, para posteriormente ser evaluada su inclusión en el libro.

Comunidad

Como se dijo anteriormente, Swift en Ejemplos es un trabajo de comunidad.

- [GitHub](#)
- [Twitter](#)
- [Equipo en Slack](#)

Listados de código

A lo largo del libro encontrarás varios listados de código donde se muestran ejemplos específicos de Swift. Hay varias consideraciones al respecto:

- Cuando el código deba de ser escrito directamente en la terminal (línea de comandos), el primer caracter en la línea será `$`. Ejemplo:

```
$ swift helloworld.swift
```

- Cuando el código deba de ser escrito en el REPL de Swift, la líneas del listado comenzarán con un número de línea. Ejemplo:

```
1> let myVar = "Hello world!"
2> print(myVar)
```

- Cuando el código deba de ser escrito en un archivo específico, se especificará el nombre del mismo como un comentario en la primera línea del listado. Ejemplo:

```
// helloworld.swift
print("Hello world!")
```

- Si se espera que la línea de código en el listado tenga algún resultado en pantalla, se denotará el mismo usando la notación `// #=>` . Ejemplo:

```
print((0...2).count) // #=> 3
```

- Los comentarios generales en las líneas de código específicas serán expresados a través de la sintaxis de comentario de Swift `//` . Ejemplo:

```
let array: [String] // Declaramos una constante de tipo Array<String>
```

Primeros Pasos

Antes de comenzar a programar con Swift, es importante preparar nuestro sistema.

Afortunadamente, Swift ahora es Open Source, y hay una versión oficial que podemos instalar en Ubuntu.

Gracias a [Swift.org](https://swift.org), el sitio oficial del lenguaje, instalar Swift en Ubuntu y OS X es trivial. Sólo debes de ir a la [página de descargas en Swift.org](#), seleccionar tu sistema, y descargar el binario precompilado.

Introducción al lenguaje

Swift es un lenguaje de programación increíble: su sintáxis es limpia, entendible (casi como pseudocódigo) sin sacrificar rendimiento y/o velocidad.

Aplicaciones

Si este es tu primer intento de aprender Swift, hay algo que debes saber: **Swift no solamente funciona para crear aplicaciones para iOS/OS X.**

Al contrario, Apple ha dicho que el objetivo de Swift es que sea un lenguaje de programación lo suficientemente simple para que sea el lenguaje con el que se enseñen a programar las nuevas generaciones, pero lo suficientemente poderoso para que puedas escribir un sistema operativo completo.

De hecho, gracias al nuevo status de *open source* de Swift, ya incluso tenemos frameworks para [crear servicios web](#) o [aplicaciones de Android](#). Ambos, usando Swift.

Tu primer programa con Swift

[Siguiendo la costumbre](#), el primer programa que escribirás en cualquier lenguaje de programación será el *Hola Mundo*.

Crea un archivo de texto `holamundo.swift` en tu escritorio y escribe lo siguiente dentro:

```
print("Hola mundo!")
```

Ahora, desde tu terminal navega a tu escritorio y escribe lo siguiente y presiona `Enter` :

```
$ swift holamundo.swift
```

Después de unos momentos verás un texto aparecer debajo de tu instrucción:

```
$ swift holamundo.swift
Hola mundo!
```

Swift incluye una función como parte del lenguaje llamada `print` que acepta un *parámetro* de tipo `String`, el cual es impreso en la pantalla cuando dicha función es llamada.

Una vez escrita esa instrucción de forma adecuada en un archivo con extensión `.swift` (la extensión que reconoce el compilador de Swift como código fuente), podemos compilarla y ejecutarla usando el comando `swift` desde nuestra línea de comandos.

Tipos de Datos

El poder de todo lenguaje nace a partir de los tipos de datos que tiene disponible. Son estos tipos de datos los que nos van a ayudar a representar ultimadamente la información y el flujo de nuestro programa.

Int

Un valor de tipo `Int` representa un número entero, que puede tener un signo positivo o negativo. Hay una variante de este tipo de dato que solamente representa números positivos: `UInt`.

`Int` también tiene variaciones específicas para representar enteros en plataformas con diferentes arquitecturas: `Int32` / `UInt32` para plataformas de 32 bits, y `Int64` / `UInt64` para plataformas de 64 bits.

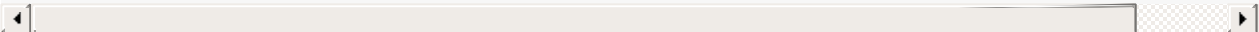
Sin embargo, no debes de preocuparte por esto, ya que:

- En plataformas de 32 bits, `Int` tendrá el mismo tamaño que `Int32`
- En plataformas de 64 bits, `Int` tendrá el mismo tamaño que `UInt64`

Lo anterior es cierto también para `UInt` y sus variantes.

Representación literal

```
let dedosEnLaMano = 5
let autosEnLaCochera = 2
let estrellasEnElCielo = 1_000_000 // Puedes usar _ entre números para mejorar la legibil
```



También puedes representar números binarios, octales y hexadecimales con lietarles en Swift:

```
let decimal = 17
let binario = 0b10001 // 17 en notación binaria
let octal = 0o21 // 17 en notación octal
let hexadecimal = 0x11 // 17 en notación hexadecimal

let suma = decimal + octal // #=> 34
```

Double

Un valor de tipo `Double` representa un número con punto flotante de 64 bits.

Los valores de tipo `Double` tienen una precisión de hasta 15 dígitos decimales, y no es necesario declarar que su tipo al momento de asignarlos a una variable o a una constante, ya que cualquier número literal con punto flotante es inferido a ser de tipo `Double`.

Representación literal

```
let decimal = 12.1875
let exponente = 1.21875e1
let hexadecimal = 0xC.3p0
```

Float

Un valor de tipo `Float` representa un número con punto flotante de 32 bits, a diferencia de los tipos de dato `Double`. Un `Float` tiene una precisión de por lo menos 6 dígitos decimales.

Representación literal

A diferencia de los tipos de dato `Double`, cuando se requiere que un tipo de dato sea tomado como `Float`, **es necesario** indicarlo a través de la anotación de tipo correspondiente:

```
let gramos: Float = 85.32
```

Variables

Una variable en Swift se declara con la palabra reservada `var`.

```
var nombre = "Oscar"
```

Las variables son conocidos como "valores mutables" dentro de Swift, no precisamente "variables."

Una de las ventajas de Swift es su excelente sistema de inferencia de tipos de datos, lo que significa que el compilador automáticamente sabe de qué tipo es tu variable sin la necesidad de expresarlo explícitamente.

El ejemplo anterior declara una variable de tipo `String`, ya que del lado derecho del signo de asignación (`=`) pusimos una representación literal de un `String`.

Si queremos declarar una variable sin asignarle un valor inicial (como en el ejemplo anterior), podemos hacerlo declarando el tipo de dato que espera la variable:

```
var nombre: String // #=> Declarar la variable nombre de tipo String sin asignar un valor  
nombre = "Oscar"
```

En ambos casos, la variable `nombre` durante el resto de la ejecución de nuestro programa, sólo podrá contener `String`s.

Una variable en Swift puede ser modificada en tiempo de ejecución, siempre y cuando se respete su tipo.

```
print(nombre) // => Oscar  
nombre = "Marco" // #=> Ahora nombre es igual a "Marco"  
print(nombre) // => Marco  
nombre = 2 // #=> Error! 2 no un tipo de dato String
```

Constantes

Una constante en Swift se declara usando la palabra reservada `let`.

```
let numeroDeDiasEnLaSemana = 7
```

Las constantes son conocidos como "valores inmutables" dentro de Swift, no precisamente "constantes."

Al igual que las variables, no es necesario declarar explícitamente el tipo de dato de la constante si se le asignará un valor inicial en la misma declaración. Podemos declarar una constante sin valor inicial, de la siguiente forma:

```
let numeroDeDiasEnLaSemana: Int // #=> Declaramos la constante tipo Int sin asignar un va
numeroDeDiasEnLaSemana = 7      // #=> Asignamos el valor a la constante
```

La característica de las constantes en Swift, es que una vez que se les asigna un primer valor, éste ya no puede ser cambiado, solamente leído.

```
numeroDeDiasEnLaSemana = 9 // Error! numeroDeDiasEnLaSemana es inmutable y ya no se puede
```

El compilador será inteligente y te sugerirá que cambies la declaración de tu constante para que ahora sea una variable (mutable) y puedas asignar ese nuevo valor.

```
$ swift constantes.swift
constantes.swift:4:24: error: immutable value 'numeroDeDiasEnLaSemana' may only be initia
numeroDeDiasEnLaSemana = 9
                        ^
constantes.swift:1:1: note: change 'let' to 'var' to make it mutable
let numeroDeDiasEnLaSemana: Int
^~~
var
```


Arrays

Un array en Swift representa una **colección de valores del mismo tipo**.

```
let enteros = [1, 2, 3]
```

En el ejemplo anterior creamos una constante llamada `enteros` y le asignamos un array de enteros usando la notación literal `[1, 2, 3]`.

Si inspeccionamos la constante `enteros` nos vamos a dar cuenta que su tipo es `Array<Int>`, o sea, array de enteros.

```
$ swift
1> let enteros = [1, 2, 3]
2> print(enteros.dynamicType)
Array<Int>
```

Esto significa que el array `enteros` solamente puede contener datos de tipo `Int`. Otra forma de representar el tipo de dato de esta constante es `[Int]`.

Para poder modificar un array después de haber sido creado, es necesario que la declaración del mismo indique que es una variable, no una constante:

```
var nombres = ["Oscar", "Marco", "Antonio"]
nombres.append("María") // #=> ["Oscar", "Marco", "Antonio", "Maria"]
nombres.removeFirst()   // #=> ["Marco", "Antonio", "Maria"]
nombres.removeAll()     // #=> []
```

En el listado de código anterior se ejecutaron una serie de métodos sobre el array, modificando su contenido.

Es importante denotar que un array (cualquier variable o constante, realmente) no puede ser manipulada o leída sino hasta después de haber sido inicializado.

```
var edades: [Int] // En este momento, se ha declarado la variable nombres, pero no ha s
print(edades.count) // La propiedad count no puede ser usada en este momento
edades = [12, 29, 22, 34]
```

Si intentamos compilar lo anterior, el compilador va a decirnos que estamos intentando usar `edades` antes de haber sido inicializada:

```
$ swift array.swift
array.swift:2:1: error: variable 'edades' used before being initialized
edades.count
^
array.swift:1:5: note: variable defined here
var edades: [Int]
    ^
```

Al invertir el orden de las líneas 2 y 3, tenemos el resultado esperado:

```
$ swift array.swift
4
```

Referencia:

1. [Documentación oficial de `Array` de Apple](#)

Funciones

Es muy fácil declarar una función en Swift. Sólo hay que usar la palabra reservada `func` :

```
func decirHola() {  
    print("hola!")  
}
```

El ejemplo en el listado anterior declara una función llamada `decirHola` , que no acepta ningún parámetro, y al ser invocada imprimirá en pantalla el texto `hola!` .

Para invocar una función hay que escribir su nombre seguido de paréntesis:

```
decirHola() // #=> hola!
```

Muchas de las funciones que escribas en un futuro trabajarán con algún tipo de información. Esta información puede ser *pasada* a la función como parámetro de la misma:

```
func sumar(primerNumero: Int, segundoNumero: Int) -> Int {  
    return primerNumero + segundoNumero  
}  
  
sumar(2, segundoNumero: 3) // => 5
```

El listado anterior declara una función que acepta dos parámetros de tipo entero (`primerNumero` y `segundoNumero`) y debe de retornar un tipo de dato entero (`-> Int`).

Primero observemos la declaración de la función.

Para Swift, dicha función se llama `sumar(_: segundoNumero:)` internamente. Observa el guión bajo en su nombre. Swift omite el nombre del primer parámetro para representar la función internamente. El nombre del segundo parámetro es usado tanto como para representar el nombre de la función internamente, como para el parámetro tal cual (para uso dentro de la función).

Esto puede hacer que el llamar una función resulte algo complicado. Como en el ejemplo anterior. ¿No sería mejor si se pudiera leer más naturalmente? La respuesta es sí:

```
func sumar(primerNumero: Int, con segundoNumero: Int) -> Int {  
    return primerNumero + segundoNumero  
}  
  
sumar(2, con: 3) // => 5
```

Ahora, nuestra función sigue teniendo dos parámetros de tipo entero. El primer parámetro se omitirá en la llamada a la función, y el segundo parámetro será representado externamente usando `con`, pero internamente usando `segundoNumero`. Estamos haciendo uso del etiquetado de parámetros.

Hacer buen uso de esta capacidad de Swift resulta en código mucho más legible, como vimos anteriormente.

Tuplas

Las tuplas agrupan datos en un mismo objeto. A diferencia de los arrays, los *miembros* de una tupla **no deben de ser del mismo tipo**.

```
let usuario = ("Oscar Swanros", 22) // => (.0 "Oscar Swanros", .1 22)
```

En el ejemplo anterior, declaramos una tupla llamada `usuario` que contiene `"Oscar Swanros"` y `22` como sus miembros. En este caso, diríamos que la tupla sería de tipo `(String, Int)`.

```
let (nombre, edad) = usuario
print(nombre)    // #=> Oscar Swanros
print(edad)      // #=> 22
```

El listado anterior muestra un ejemplo de lo que llamamos *pattern matching*. En una misma línea, utilizamos la sintaxis `let (nombre, edad)` para recuperar los datos en la tupla `usuario`. Los contenidos de la tupla se *mapean* a las constantes definidas: el primer miembro de la tupla se asigna a la primera constante, el segundo a la segunda, etc.

Usando un guión bajo, podemos ignorar uno o más miembros de la tupla al momento de recuperarlos:

```
let (_, edad) = usuario
print(edad)      // #=> 22
```

Podemos también hacer referencia a miembros de la tupla sin necesidad de recuperarlos y asignarlos a constantes o variables externas:

```
print(usuario.0)    // #=> Oscar Swanros
print(usuario.1)    // #=> 22
```

Al igual que con las funciones y sus parámetros etiquetados, podemos poner etiquitas a los miembros de una tupla:

```
let usuario = (nombre: "Oscar Swanros", edad: 22) // => (.0 "Oscar Swanros", .1 22)
usuario.nombre    // => Oscar Swanros
usuario.edad      // => 22
```

Aunque una "tupla" no tiene un tipo único en Swift como `Int`, `Array` o `String`, podemos usar las tuplas como cualquier otro tipo de dato en nuestros programas. Incluso como tipo de retorno en funciones:

```
func generarUsuario(nombre nombre: String, edad: Int) -> (nombre: String, edad: Int) {  
    return (nombre, edad)  
}  
  
let usuario = generarUsuario(nombre: "Oscar Swanros", edad: 22)  
print(usuario.nombre)    // #=> Oscar Swanros  
print(usuario.edad)      // #=> 22
```

Es importante diferenciar las tuplas de los arrays.

- **Un array tiene elementos**, puede ser manipulado, escrito y leído.
- **Una tupla tiene miembros**, solamente pueden ser leídos y reemplazados de forma individual, siempre y cuando no se afecte el tipo de dato de la tupla.

Swift Intermedio

En este punto ya conoces los conceptos básicos del lenguaje. Estás familiarizado con la sintaxis básica de los términos que se usan para crear programas.

En esta sección, aprenderás técnicas para hacer que tu código sea mucho más productivo.

Control de flujo

Uno de los aspectos más importantes de cualquier lenguaje de programación es la habilidad de poder tomar decisiones sobre el flujo de la ejecución del mismo en base a los datos con los que se está trabajando.

Swift ofrece varias alternativas para lograr lo anterior.

if / else

La forma más básica de poder decidir hacer algo o no en Swift es usando los bloques `if` y `else` .

```
if numeroDeInvitados > 10 {  
    print("No caben en mi casa!")  
} else {  
    print("Bienvenidos")  
}
```

Las llaves son **requeridas** en cada uno de estos bloques. Puedes escribir un `if` sin un `else` , pero no un `else` sin un `if` .

Inmediatamente después el `if` , escribe una expresión que retorne un valor `Bool` . Si el resultado de esta expresión es `true` , se ejecutará el bloque `if` . De lo contrario, se ejecutará el bloque `else` , en caso de existir.

```
if true {  
    print("Siempre se ejecutará")  
}
```

En caso de que necesites hacer más de una validación, puedes incluir un bloque `else if` :

```
if esSabado {  
    comprarDespensa()  
} else if esDomingo {  
    noLevantarseDeCama()  
} else {  
    trabajar()  
}
```

En listado de código anterior nos dice qué deberíamos hacer dependiendo del día de la semana. En sábado, iremos a comprar la despensa, en domingo no nos levantaremos de la cama. Desafortunadamente, cualquier otro día de la semana debemos ir a trabajar.

switch

guard

Swift 2.0 introdujo la palabra reservada `guard`.

```
func esPositivo(numero: Int) -> Bool {  
    guard numero > 0 else {  
        return false  
    }  
  
    return true  
}
```

Un `guard` nos permite realizar chequeos en el flujo de nuestro programa. Es como decir "para que el programa siga su ejecución, necesito que esta condición sea validada, de lo contrario, ejecuta el código en la cláusula `else`."

Es importante notar que el código dentro de la cláusula `else` **debe** de cambiar el flujo de ejecución del programa. Puedes usar cualquiera de estas palabras reservadas para cambiar el flujo del programa desde una cláusula `else` en un `guard`:

- `return`
- `break`
- `continue`
- `throw`

O en su defecto, llamar a una función marcada con el atributo `@noreturn`.

```
func imprimir(cadena: String?) {  
    guard let cadena = cadena else {  
        return  
    }  
  
    print(cadena)  
}
```

En el ejemplo anterior, usamos `guard` para asegurarnos de que el opcional `cadena` tenga un valor.

- Si esa condición se cumple, el valor de `cadena` se transferirá a la constante `cadena` (que a partir de ahí será de tipo `String`, no `String?`), y continúa la ejecución normal del programa.
- Si no se cumple la condición, se ejecutará el bloque de código en `else`, que en este caso simplemente retorna la función.

Casos de Uso

Imagina una pieza de código donde se necesita evaluar diferentes condiciones para que el programa pueda continuar. Antes de Swift 2.0, ese código se vería algo así:

```
typealias Usuario = (nombre: String, apellido: String, edad: Int, direccion: String)

func generarUsuario(nombre: String?, apellido: String?, edad: Int?, direccion: String?) {
    if let nombre = nombre {
        if let apellido = apellido {
            if let edad = edad {
                if let direccion = direccion {
                    return (nombre, apellido, edad, direccion)
                }
            }
        }
    }

    // Si el parámetro nombre era nil, no podemos construir la tupla Usuario
    return nil
}
```

Usando `guard`, nuestro código cambia tremendamente:

```
func generarUsuario(nombre: String?, apellido: String?, edad: Int?, direccion: String?) {
    guard
        let nombre = nombre,
        let apellido = apellido,
        let edad = edad,
        let direccion = direccion else {
        return nil
    }

    return (nombre, apellido, edad, direccion)
}
```

where

La palabra reservada `where` nos permite ser aún más precisos al momento de realizar verificaciones en el flujo de nuestro programa.

Un ejemplo claro de esto es:

```
let nombres = ["Oscar", "Marco", "Yuri"]

for nombre in nombres where nombre == "Oscar" {
    print(nombre) // #=> Oscar
}
```

En el ejemplo anterior:

- Iteramos cada elemento del array `nombres`
- El elemento en turno es asignado a `nombre`
- Se ejecutará el código entre llaves solamente si la condición después de `where` se cumple

```
var ceros: [Int]? = []

while let nums = ceros where nums.count < 5 {
    ceros?.append(0) // [0,0,0,0,0]
}
```

Ahora combinamos un ciclo `while`, donde su condicional incluye un `where`. Otra aplicación bastante útil es en los condicionales `if` y en conjunto con `guard` s:

```
func darAcceso(nombreCompleto: String?) -> Bool {
    if let nombre = nombreCompleto where nombre == "Oscar Swanros" {
        return true
    }

    return false
}

darAcceso("Oscar Swanros") // #=> true

func verificarMayoriaDeEdad(nombreCompleto: String?, edad: Int) -> String {
    guard let nombre = nombreCompleto where edad >= 18 && nombre == "Oscar Swanros" else
        return "Lo siento, no puedes entrar."
    }

    return "Bienvenido al club, \(nombre)"
}

verificarMayoriaDeEdad("Oscar Swanros", edad: 14) // #=> Lo siento, no puedes entrar
```

Otros usos de `where` :

Switch:

```
let tupla = (14, "Gerardo Swanros")

switch tupla {
case let (edad, _) where edad >= 18:
    print("Eres mayor de edad!")

case let (_, nombre) where nombre == "Oscar Swanros":
    print("Eres \(nombre)!")

default:
    print("No se nada sobre ti")
}

// #=> No se nada sobre ti
```

Restricciones de tipo en funciones genéricas

```
enum Animal {  
    case Perro  
}  
  
extension Animal: CustomStringConvertible {  
    var description: String {  
        switch self {  
            case .Perro: return "El mejor amigo del hombre"  
        }  
    }  
}  
  
func obtenerDescripcionDeObjeto<T where T: CustomStringConvertible>(objeto: T) -> String  
    return objeto.description  
}  
  
obtenerDescripcionDeObjeto(Animal.Perro) // #=> El mejor amigo del hombre
```

Enumeraciones

Las enumeraciones en Swift comprenden una de sus características más poderosas. Nos permiten representar información de una manera mucho más clara. Se declaran usando la palabra reservada `enum` :

```
enum TipoDeUsuario {  
    case Admin  
    case Editor  
    case Autor  
    case Lector  
}
```

El listado de código anterior define una enumeración llamada `TipoDeUsuario` que tiene como opciones `.Admin` , `.Editor` , `.Autor` y `.Lector` . En este contexto, nuestro programa ahora cuenta con un nuevo tipo de dato `TipoDeUsuario` y podemos usarlo al igual que usaríamos cualquier `Int` ó `String` .

Podemos crear una *instancia* de una enumeración de la siguiente manera:

```
let admin = TipoDeUsuario.Admin  
let autor: TipoDeUsuario = .Autor
```

enums con tipos de dato de respaldo

Ambas constantes son de tipo `TipoDeUsuario` .

Las enumeraciones también se pueden crear a partir de datos. A continuación declaramos una enumeración que tiene `String` como tipo de dato de respaldo para cada caso:

```
enum UtencilioDeCocina: String {  
    case Tenedor = "tenedor"  
    case Cuchillo = "cuchillo"  
    case Cuchara = "cuchara"  
}  
  
let tenedor = UtencilioDeCocina(rawValue: "tenedor") // => Tenedor  
let cuchillo = UtencilioDeCocina(rawValue: "cuchara sopera") // => nil, "cuchara sopera"
```


Usando el constructor `rawValue:` podemos pasar un argumento del tipo de dato que respalda a la enumeración. Si el dato que pasamos al constructor es igual a uno de los declarados en la enumeración, se generará una instancia de la enumeración.

Del mismo modo, podemos obtener el valor de respaldo de una instancia de una enumeración usando la propiedad `rawValue` :

```
print(UtencilioDeCocina.Cuchillo.rawValue) // #=> "cuchillo"
```

enums con tipos de datos asociados

Las enumeraciones, además poder tener tipos de dato de respaldo, pueden, por así decirlo, transportar datos.

```
enum Resultado {  
    case Error(Int, String)  
    case Valor(AnyObject)  
}
```

En el listado de código anterior definimos una enumeración con dos `cases` que pueden transportar datos: `.Error` que tiene asociados un `Int` y un `String`, y `.Valor`, que puede transportar cualquier objeto.

```
let error = Resultado.Error(404, "Página no encontrada")  
  
switch error {  
case .Error(let codigo, let descripcion):  
    print("Código \(codigo): " + descripcion)  
  
case .Valor(let valor):  
    print(valor)  
}  
  
// #=> "Código 404: Página no encontrada"
```

El listado anterior muestra cómo podemos trabajar cuando tenemos enumeraciones con datos asociados.

Opcionales

Si Swift tiene algo que llama la atención a primera vista, son los signos de interrogación y exclamación que se llegan a ver en el código de vez en vez.

```
enum NivelDeEstudios {  
    case Primaria  
    case Secundaria  
    case Preparatoria  
}  
  
var nivelDeEstudios: NivelDeEstudios?
```

A cualquier tipo de dato que sea seguido de un signo de interrogación se le conoce como un valor opcional.

En el ejemplo anterior, tenemos un valor `NivelDeEstudios?` (se lee "Nivel de estudios opcional"). Esto significa que la variable `nivelDeEstudios` **puede** contener un valor de tipo `NivelDeEstudios`, pero **no es seguro que lo contenga**. Es como si la variable dijera: "tal vez tenga un string dentro, no sé."

Los opcionales en Swift nos ayudan a manejar de mejor manera la información con la que trabaja nuestro programa: en un programa que recabe datos para una encuesta, el encuestado tal vez quiera o no decir su nivel de estudios, es por eso que esa variable está marcada como opcional.

```
var nivelDeEstudios: NivelDeEstudios? = .Preparatoria  
  
func terminarEncuesta() {  
    // No sabemos si nivelDeEstudios contiene un valor  
    // Si sí contiene un valor, se lo asignamos a  
    // la constante nivel.  
    if let nivel = nivelDeEstudios {  
        // En este momento, nivel tiene el valor de nivelDeEstudios  
        // pero nivel es de tipo String, no de tipo String?  
        guardar(nivel)  
    }  
}
```

En el listado de código anterior sucede lo siguiente:

1. Declaramos la variable `nivelDeEstudios` como un `NivelDeEstudios?`
2. El programa sigue su ejecución.

3. Al momento en que la ejecución del programa llega a la función `terminarEncuesta`, debemos de verificar si esa variable tiene o no datos.
4. Usamos **Extracción de Opcionales** (u **Optional Binding**, en Inglés), para que, en caso de que la variable `nivelDeEstudios` **contenga** algún valor, éste sea asignado a la constante `nivel`.
5. La variable `nivel` ahora es un valor de tipo `NivelDeEstudios` que podemos manipular libremente, porque sabemos que es un valor que existe.

Conocer bien los opcionales es importante, puesto que nos protegen de errores en nuestros programas donde podríamos creer que tenemos información para procesar, pero en realidad no es así.

Cuidado

Existe otra forma de obtener los valores de una variable o constante declarada como opcional, y es a través de la **extracción forzada de opcionales**. Es aquí donde el signo de exclamación entra en juego.

```
var nivelDeEstudios: NivelDeEstudios? = .Preparatoria

func terminarEncuesta() {
    // No verificamos que nivelDeEstudios tenga un valor real antes de intentar usarlo
    guardar(nivelDeEstudios!)
}
```

El listado anterior muestra un escenario donde no se verifica que el contenido de un opcional exista. Sí, el código puede llegar a verse más limpio, pero también **introduce un gran riesgo**.

¿Qué pasa cuando quiero trabajar con un valor que no existe? Simple, nuestro programa va a detenerse abruptamente, puesto que quisimos usar un espacio de memoria que no existía. **Eso es un error gravísimo, y debemos evitarlo a toda costa**. Es por eso, después de todo, que existe el concepto de "valor opcional" en primer lugar.

En pocas palabras: no uses nunca la extracción forzada de opcionales. Es muy mala práctica, y es peligroso.

Estructuras

Las estructuras (`struct`) son uno de los pilares principales de Swift como lenguaje.

```
struct Vaso {  
    let contenido: String  
    let capacidad: Float  
}
```

El listado de código anterior define una estructura llamada `Vaso` que tiene dos *propiedades*:

1. `contenido` , de tipo `String`
2. `capacidad` , de tipo `Float`

Las estructuras representan unidades de información. A diferencia de las clases (que se discutirán en el capítulo siguiente), lo que importa con las estructuras en Swift es su contenido.

Si una estructura (A) tiene los mismos valores en sus propiedades que otra estructura (B), se considera que A y B son iguales. Por lo tanto, son intercambiables.

```
let vaso1 = Vaso(contenido: "Agua", capacidad: 350)  
let vaso2 = Vaso(contenido: "Leche", capacidad: 350)  
let vaso3 = Vaso(contenido: "Agua", capacidad: 350)  
  
if vaso1 == vaso2 {  
    print("El primer y el segundo vaso no son iguales")  
} else if vaso1 == vaso3 {  
    print("El primer y el tercer vaso son iguales")  
} else {  
    print("Nope")  
}
```

Si ejecutáramos el código anterior veríamos impresa en pantalla la leyenda "El primer y el tercer vaso son iguales."

Aunque realmente el espacio de memoria que ocupan `vaso1` y `vaso2` son diferentes, se considera que son iguales puesto que su contenido es idéntico.

Constructores

Toda estructura con propiedades definidas tiene un constructor por defecto que requiere que se le asigne un valor a cada una de las propiedades. Si alguna de las propiedades tiene un valor inicial, dicha propiedad se omite en el constructor por defecto:

```
struct Animal {
    let numeroDePatas = 4
    let familia: FamiliaAnimal
}

let animal = Animal(familia: .Felidae) // => Animal {numeroDePatas: 4, familia: Felidae }
```

Modificando una estructura

Las estructuras con propiedades declaradas como valores inmutables `let` es por defecto inmutable por si misma.

```
struct Billete {
    let valor: Float
    let color: String
}

let miBillete = Billete(valor: 200, color: "#68C3A3")
miBillete.valor // #=> 200
miBillete.valor = 500 // #=> Error!
```

Si intentas compilar lo anterior, encontrarás el error:

```
$ swift struct.swift
struct.swift:8:17: error: cannot assign to property: 'valor' is a 'let' constant
miBillete.valor = 500 // #=> Error!
~~~~~ ^
struct.swift:2:5: note: change 'let' to 'var' to make it mutable
    let valor: Float
    ^~~
    var
```

Cambiar la declaración de una propiedad de `let` a `var` en la definición de una estructura hará que esa propiedad pueda ser modificada en tiempo de ejecución.

Puesto que las estructuras en Swift son consideradas como entidades *de valor*, si deseas modificar una instancia de una estructura, tendrás que declarar que el valor es mutable al momento de crear la misma.

Las propiedades declaradas como valores mutables (`var`) serán parte del constructor por defecto aún cuando tengan un valor inicial declarado.

```
struct Billete {
    var valor: Float = 200
    var color: String
}

var miBillete = Billete(valor: 200, color: "#68C3A3")
miBillete.valor // #=> 200
miBillete.valor = 500 // #=> 500
```

Observa cómo ahora `miBillete` es un valor mutable (`var`), lo que significa que podemos modificar su contenido.

Aunque el precio de tener una estructura mutable puede parecer mínimo, el punto general de las estructuras en Swift es que sean valores inmutables. Es ampliamente recomendado que se respete este principio.

Ejemplo práctico: un billete no puede cambiar su valor después de haber sido impreso. Puedes tener dos billetes iguales, del mismo valor, del mismo color, y puedes intercambiarlos por otros billetes del mismo valor. Al final de cuenta, te importa cuánto vale, no un billete en específico.

Métodos

Hasta ahora hemos visto solamente cómo una estructura puede representar un valor, pero no hemos hablado de cómo una estructura puede interactuar con nuestro programa.

```
struct Persona {
    let nombre: String
    let apellido: String

    func nombreCompleto() -> String {
        return nombre + " " + apellido
    }
}

let oscar = Persona(nombre: "Oscar", apellido: "Swanros")
oscar.nombreCompleto() // #=> "Oscar Swanros"
```

En el ejemplo pasado, definimos una estructura llamada `Persona` con dos propiedades y un método. (Se llama "método" y no "función" puesto que se encuentra declarado como parte de la definición de un tipo de dato nuevo.)

Dentro de nuestro método podemos usar las propiedades de nuestra estructura. Por ahora, sólo estamos leyendo los valores `nombre` y `apellido` y los estamos concatenando. Pero ¿qué pasaría si quisiéramos un método que modifique los valores en nuestra estructura? Hay que especificarlo:

```
struct Persona {  
    var nombre: String  
    let apellido: String  
  
    func nombreCompleto() -> String {  
        return nombre + " " + apellido  
    }  
  
    mutating func actualizarNombre(nuevoNombre: String) {  
        nombre = nuevoNombre  
    }  
}  
  
var mutante = Persona(nombre: "Oscar", apellido: "Xavier")  
mutante.nombreCompleto()           // #=> "Oscar Xavier"  
mutante.actualizarNombre("Charles") // #=> "Charles Xavier"
```

Preceder la declaración de un método con la palabra reservada `mutating` indica que dicho método, al ser ejecutado, modificará el valor sobre el que actúa (en este caso, una instancia de la estructura `Persona`). Aún con esa declaración, es necesario que tanto la propiedad a mutar como la instancia del valor sean declarados como valores mutables (`var`).

Llamar un método en una estructura basta con solo poner un punto después de la instancia y llamar al método con los parámetros requeridos.