El Lenguaje de Programación Swift

Alsey Coleman Miller

Published with GitBook

Tabla de contenido

El Lenguaje de Programación Swift

Este libro es una traduccion de "*The Swift Programming Language*", que se encuentra en swift.org.

Links

- swiftespanol.com
- GitBooks
- GitHub

Bienvenidos a Swift

Sobre Swift

Swift es un nuevo lenguaje de programación para aplicaciones de iOS, OS X, watchOS, y tvOS que está cimentado en lo mejor de C) y Objective-C, sin las restricciones de compatibilidad con C. Swift adopta patrones de programación seguros y añade características modernas para hacer el proceso más fácil, más flexible y más divertido. La implementación nueva y limpia de Swift, apoyado por los marcos de trabajo de Cocoa y Cocoa Touch dan una oportunidad para re-imaginar como funciona el desarrollo de software.

El proceso de la creación de Swift se ha dado por varios años. Apple creó el cimiento para este, mejorando nuestra infraestructura de compilador, depurador y marcos de trabajo. Hemos simplificado el manejo de la memoria gracias a la creación de Conteo de Referencia Automatica. Nuestros marcos de trabajo, construidas sobre la base sólida de Foundation y Cocoa, han sido modernizadas y estandarizadas. Objective-C, por sí mismo, ha evolucionado para soportar clausuras, colecciones literales y módulos, habilitando a que los marcos de trabajo adopten tecnologías de lenguas modernas sin obstáculos. Gracias a este trabajo, podemos introducir un nuevo lenguaje de desarrollo para el futuro de desarrollo de software para las plataformas de Apple.

Swift se siente familiar para desarrolladores de Objective-C. Adopta la legibilidad de parámetros nombrados provenientes de Objective-C, así como el modelo de objectos dinámicos. Provée acceso sin interrupciones a marcos de trabajo existentes de Cocoa y completa interoperabilidad con Objective-C. Construyendo a partir de esta fundación, Swift introduce muchos nuevos conceptos y unifica los aspectos procedurales y orientado a objetos del lenguaje.

Swift es amigable para nuevos programadores. Es el primer lenguaje de programación de sistemas que cuenta con calidad industrial y que, a su vez, es tan expresivo y agradable como un lenguaje interpretado. Soporta *playgrounds*, una característica innovadora que da paso a la experimentación con el código Swift y permite ver los resultados inmediatamente, sin tener que compilar y ejecutar la aplicación.

Swift combina la mejor filosofía de los lenguajes modernos con la sabiduría de la amplia cultura en ingeniería de Apple. El compilador está optimizado para el mejor rendimiento, y el lenguaje para un óptimo desarrollo, sin comprometer ninguno de los dos. Está diseñado para escalar de la implementación de `hola mundo` hasta un sistema operativo entero. Todo esto hace que Swift sea una inversión segura para desarrolladores y Apple.

Así mismo, es una manera fantástica para escribir aplicaciones iOS, OS X, watchOS, y tvOS, y continuará evolucionándose con nuevas características y nuevas capacidades. Nuestras metas para Swift son ambiciosas. No podemos esperar para ver qué podrás crear con ello.

Una Guia de Swift

La tradición sugiere que el primer programa en un nuevo lenguaje de desarrollo debería imprimir las palabras "Hola, mundo!" en la pantalla. En Swift, esto se puede hacer con una sola línea de código:

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

Si has escrito código en C o en Objective-C, esta sintaxis debería serte familiar. En Swift, esta línea de código es un programa completo. No necesitas importar una librería para funcionalidad de entrada y salida o manejo de texto. El código escrito en el contexto global es usado como un punto de entrada para la aplicación, entonces no necesitas una función principal o `main()`, así como tampoco lo es el escribir punto y coma al final de cada sentencia.

Esta guía te da suficiente información para empezar a escribir código en Swift a partir de la muestra del cómo implementar una variedad de tareas de programación. No te preocupes si no entendiste algo, todo lo aquí mencionado está explicado a detalle en el resto del libro.

Valores Simples

Se usa `let` para crear una constante y `var` para crear una variable. El valor de una constante no necesita saberse a la hora de compilar, pero hay que asignarle un valor exactamente una vez. Esto significa que puedes usar constantes para nombrar un valor que se determina una vez y usarlo en muchos lugares.

```
var miVariable = 42
miVariable = 50
let miConstante = 42
```

Una constante o variable debería tener el mismo tipo de valor que le quieras asignar. Sin embargo, no necesitas escribirlo explícitamente todo el tiempo. Proveer un valor cuando creas la constante o variable, permite que el compilador deduzca su tipo. En el ejemplo de arriba, el compilador deduce que `miVariable` es un número entero por que su valor inicial también lo es.

Si el valor inicial no provée suficiente información (o si no hay un valor inicial), especifica el tipo escribiéndolo después de la variable, separado por dos puntos como se ve a continuación.

```
let enteroImplicito = 70
let dobleImplicito = 70.0
let dobleExplicita: Double = 70
```

> ## Experimento
>
> Crea una constante con un tipo explícito de `Float` y un valor de `4`.

Los valores nunca son implícitamente convertidos a otro tipo. Si necesitas convertir un valor a otro tipo, explícitamente haz una instancia del tipo deseado.

```
let etiqueta: "El ancho es "
let ancho = 94
let etiquetaAncho = etiqueta + String(ancho)
```

> ## Experimento
>
> Intenta remover la conversión a `String` de la última línea. ¿Qué error te aparece?

Hay una manera más simple de incluir varios valores en el texto: Escribe el valor entre paréntesis, y escribe una barra invertida (`\`) antes de estos. Por ejemplo:

```
let manzanas = 3
let naranjas = 5
let manazanaResumen = "Yo tengo \(manzanas) manzanas."
let frutaResumen = "Yo tengo \(manzanas + naranjas) pedazos de fruta."
```

> ## Experimento
>
> Usa `\()` para incluir un cálculo de coma flotante en un texto y para incluir el nombre de alguien en un saludo.

Crea arreglos) y diccionarios usando corchetes (`[]`) y accede sus elementos escribiendo el índice o llave en corchetes. Una coma se permite después del primer elemento.

```
var miListaDeCompras = ["bagre", "agua", "tulipanes", "pintura azul"]
miListaDeCompras[1] = "botella de agua"

var profesiones = [
    "Malcolm": "Capitán",
    "Kaylee": "Mecánico"
]

profesiones["Jayne"] = "Relaciones Públicas"
```

Si la información de tipo puede ser deducida, puedes escribir un arreglo vacío como `[]` y un diccionario vacío como `[:]` : Esto podría ocurrir, por ejemplo, cuando creas un nuevo valor para una variable o pasas un de argumento a una función.

```
miListaDeCompras = []
profesiones = [:]
```

Control de Flujo

Usa `if` y `switch` para crear sentencias condicionales, y usa `for-in` , `for` , `while` , y `repeat` para hacer un ciclo. Los paréntesis cerrando la condición o ciclo son opcionales, pero las llaves sobre el cuerpo son requeridos.

```
let puntajesIndividuales = [75, 43, 103, 87, 12]

var puntajeDeEquipo = 0

for puntaje in puntajesIndividuales {

    if puntaje > 50 {
        puntajeDeEquipo += 3
    } else {
        puntajeDeEquipo += 1
    }
}
print(puntajeDeEquipo)
```

En una sentencia de `if` , la sentencia condicional debería ser una expresión booleana, lo cual significa que un código como `if score { ... }` es un error mas no una comparación implícita a cero. Puedes usar `if` y `let` juntos para trabajar con valores que podrían estar ausentes. Estos valores se representan como opcionales. Un valor opcional contiene un valor `nil` (nulo) para representar que un valor esta faltando. Escribe un signo de interrogación (`?`) después del tipo de valor para marcarlo como opcional.

```
var textoOpcional: String? = "Hello"
print(textoOpcional == nil)

var nombreOpcional: String? = "John Appleseed"
var saludo = "Hello!"
if let nombre = nombreOpcional {
    saludo = "Hello, \(name)"
}
```

> ## Experimento
>
> Cambia `textoOpcional` a `nil`. ¿Qué saludo se obtiene? Añade una cláusula de `else` que asigna otro saludo si `nombreOpcional` es `nil`.

Si el valor opcional es `nil`, el valor condicional es falso y el código en las llaves se saltea. De lo contrario, el valor opcional es desenvuelto y asignado al constante después de `let`, lo cual hace al valor desenvuelto disponible dentro del bloque de código.

Otra manera de manejar valores opcionales es proveer un valor predeterminado usando el operador `??`. Si el valor opcional está faltando, el valor predeterminado es usado en su lugar.

```
let apodo: String? = nil
let nombreCompleto: String = "John Appleseed"
let saludoInformal = "Hola \(apodo ?? nombreCompleto)"
```

`switch` soporta cualquier tipo de datos y una amplia variedad de operaciones de comparación: no están limitados a números enteros y pruebas de igualdad.

```
let vegetal = "ají rojo"
switch vegetal {
case "apio":
    print("Añade unas pasas y prepara un bocado.")
case "pepino", "berro":
    print("Eso haría un buen plato.")
case let x where x.hasPrefix("ají")
    print("Es un \(x) picante?")
default:
    print("Todo sabe bien en sopa.")
}
```

> ## Experimento
>
> Intenta remover el caso predeterminado (`default:`). ¿Que error recibes?

Nota como `let` puede ser usado en un patron para asignar el valor que emparejó esa parte del patron a un constante.

Después de ejecutar el código dentro del caso que emparejo, el programa sale del sentencia `switch`. Ejecución no continua al siguiente caso, entonces no hay necesidad de explícitamente cesar (con `break`) cada caso.

Puedes usar `for-in` para iterar sobre elementos en un diccionario proveyendo una pareja de nombres que se usara para cada pareja de llave y valor (*key-value* en inglés). Diccionarios son una colección sin orden, entonces sus llaves y valores son iterados en una orden arbitraria.

```
let numerosInteresantes = [
    "Primo": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Cuadrado": [1, 4, 9, 16, 25],
]
var masGrande = 0
for (tipo, numeros) in numerosInteresantes {
    for numero in numeros {
        if numero > masGrande {
            masGrande = numero
        }
    }
}
print(masGrande)”
```

> ## Experimento
>
> Añade otra variable para registrar que tipo de número era el mas grande, así también como su valor.

Usa `while` para repetir un bloque de código hasta que una condición cambie. La condición de un ciclo puede estar al final también, asegurando que el ciclo se ejecute una vez a lo mínimo.

```
var n = 2
while n < 100 {
    n = n * 2
}
print(n)

var m = 2
repeat {
    m = m * 2
} while m < 100
print(m)
```

Puedes guardar una indice en una ciclo, usando `..<` para hacer un rango de índices, o mediante inicialización explicita, condición, e incremento. Estos dos ciclos hacen lo mismo:

```
var primerCiclo = 0
for i in 0..<4 {
    primerCiclo += i
}
print(primerCiclo)

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

Funciones y Clausuras

Usa `func` para declarar una función. Llama una función anexando una lista de argumentos en paréntesis a la nombre de la función. Usa `->` para separar los nombres de los parámetros y tipos del tipo de valor de retorno de la función.

```
func saludar(nombre: String, dia: String) -> String {
    return "Hola \(nombre), hoy es \(dia)."
}

saludar("Bob", dia: "Martes")
```

> ### Experimento
>
> Remueve el parámetro `dia`. Añade un parámetro que incluya el almuerzo de hoy en el saludo.

Usa una tupla para hacer un valor compuesta: por ejemplo, para retornar múltiples valores de una función. Los elementos de una tupla se pueden referir por nombre o número.

```
func calcularEstadisticas(puntajes: [Int]) -> (min: Int, max: Int, sum: Int) {
    var min = scores[0]
    var max = scores[0]
    var sum = 0

    for puntaje in puntajes {
        if puntaje > max {
            max = puntaje
        } else if puntaje < min {
            min = puntaje
        }
        sum += puntaje
    }

    return (min, max, sum)
}
let estadisticas = calcularEstadisticas([5, 3, 100, 3, 9])
print(estadisticas.sum)
print(estadisticas.2)
```

Funciones también pueden tomar una cantidad variada de argumentos, collecionandolos en un arreglo.

```
func suma(numeros: Int...) -> Int {
    var suma = 0
    for numero in numeros {
        suma += numero
    }
    return suma
}
suma()
suma(42, 597, 12)
```

> ## Experimento
>
> Escribe una función que calcula el valor promedio de sus argumentos.

Funciones pueden ser anidados. Funciones anidados tienen acceso a variables que fueron declaradas en la función exterior. Puedes usar funciones anidados para organizar el código en una función que es larga o compleja.

```
func retornarQuince() -> {
    var y = 10
    func añadir() {
        y += 5
    }
    añadir()
    return y
}
retornarQuince()
```

Funciones son un tipo de primera clase. Esto significa que una función puede retornar otra función como su valor.

```
func crearIncrementador() -> ((Int) -> (Int)) {
    func añadeUno(numero: Int) -> Int {
        return 1 + numero
    }
    return añadeUno
}
var incrementa = crearIncrementador()
incrementa(7)
```

Una función puede tomar otra función como uno de sus argumentos.

```
func tieneAlgunPareja(lista: [Int], condicion: (Int -> Bool)) -> Bool {
    for elemento in lista {
        if condicion(elemento) {
            return true
        }
    }
    return false
}

func menosQueDiez(numero: Int) -> Bool {
    return numero < 10
}
var numeros = [20, 19, 7, 12]
tieneAlgunPareja(numeros, condicion: menosQueDiez)
```

Funciones en verdad son un caso especial de clausuras: bloques de código que pueden ser llamados después. El código en una clausura tiene acceso a cosas como variables y funciones que estaban disponibles en el contexto donde fue creado, aun si la clausura esta siendo ejecutando en otro contexto: ya viste un ejemplo de esto con funciones anidados. Puedes escribir una clausura sin nombre via encerrando código con llaves (`{}`). Usa `in` para separar los argumentos y tipo de valor retornado del cuerpo de la función.

```
numeros.map({ (numero: Int) -> Int in
    let resultado = 3 * numero
    return resultado
})
```

> ## Experimento
>
> Vuelve a escribir la clausura para que retorne cero para todo número impar.

Tienes varias opciones para escribir clausuras mas concisamente. Cuando ya se sabe el tipo de clausura, así como una retrollamada) para un delegado), puedes omitir el tipo de los parámetros, valor de retorno o ambos. Clausuras de una sola sentencia implícitamente retornan el valor de su única sentencia.

```
let numerosMapeados = numeros.map({ numero in 3 * numero })
print(numerosMapeados)
```

Puedes referir a parámetros por número envés de su nombre, esta estrategia es especialmente útil en clausuras cortos. Una clausuras pasado como un ultimo argumento puede aparecer inmediatamente después de los paréntesis. Cuando una clausura es el único argumento a una función, puedes omitir los paréntesis completamente.

```
let numerosOrdenados = numeros.sort { 0$ > $1 }
print(numerosOrdenados)
```

Objetos y Clases

Usa `class` seguido del nombre de la clase para crear una clase. Una declaración de una propiedad en una clase es escrita de la misma manera que la declaración de una constante o variable, con la excepción de que esta en el contexto de la clase. Declaraciones de métodos y funciones están escritas de la misma manera.

```
class Figura {
    var numeroDeLados = 0
    func descripcionSimple() -> String {
        return "Una figura con \(numeroDeLados) lados."
    }
}
```

> ## Experimento
>
> Añade una propiedad constante con `let` y añade otra propiedad que toma un argumento.

Crea una instancia de una clase poniendo paréntesis después del nombre de la clase. Usa puntos (`.`) para acceder los propiedades y métodos de la instancia.

```
var figura = Figura()
figura.numeroDeLados = 7
var descripcionDeFigura = figura.descripcionSimple()
```

A esta version de la clase `Figura` le esta faltando algo importante: un inicializador para configurar la clase cuando una instancia se crea. Usa `init` para crear uno.

```
class FiguraNombrado {
    var numeroDeLados: Int = 0
    var nombre: String

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

    func descripcionSimple() -> String {
        return "Una figura con \(numeroDeLados) lados."
    }
}
```

Nota como `self` es usado para distinguir la propiedad `nombre` del argumento al inicializador también llamado `nombre` . Sus argumentos al inicializador son pasados como una llamada de función cuando creas una instancia de la clase. Toda propiedad necesita un valor asignado, sea en su declaración (como con `numeroDeLados`) o en el inicializador (como con `nombre`).

Usa `deinit` para crear el deinicializador si necesitas hacer operaciones de limpieza cuando el objeto es deallocado.

Subclases incluyen el nombre de su superclase (o clase padre) después del nombre de su clase, separado por un colon. No hay requerimiento para que clases sean el subclase de alguna clase principal, entonces puede incluir o omitir una superclase según tus necesidades.

Métodos en una subclase que redefinen la implementación de la superclase están marcados con `override` . Redefinir un método por accidente, sin marcar `override` , es detectado por el compilador como un error. El compilador también detecta métodos con

`override` que en verdad no redefinen ningún método en su superclase.

```
class Cuadrado: FiguraNombrado {
    var anchoDeLado: Double

    init(anchoDeLado: Double, nombre: String) {
        self.anchoDeLado = anchoDeLado
        super.init(nombre: nombre)
        numeroDeLados = 4
    }

    func area() ->  Double {
        return anchoDeLado * anchoDeLado
    }

    override func descripcionSimple() -> String {
        return "Un cuadrado con \(anchoDeLado) ancho de lados."
    }
}
let prueba = Cuadrado(anchoDeLado: 5.2, nombre: "mi cuadrado de prueba")
prueba.area()
prueba.descripcionSimple()
```

> ## Experimento
>
> Haz otra subclase de `FiguraNombrado` llamado `Circulo` que toma una radio y un nombre como argumentos a su inicializador. Implementa los métodos `area()` y `descripcionSimple()` en la clase `Circulo`.

Además de propiedades simples que son guardados, propiedades pueden tener métodos consultores (`get`) y modificadores (`set`).

```
class TrianguloEquilateral: FiguraNombrado {
    var anchoDeLado: Double = 0.0

    init(anchoDeLado: Double, nombre: String) {
        self.anchoDeLado = anchoDeLado
        super.init(nombre: nombre)
        numeroDeLados = 4
    }

    var perimetro: Double {
        get {
            return 3.0 * anchoDeLado
        }
        set {
            anchoDeLado = newValue / 3.0
        }
    }

    override func descripcionSimple() -> String {
        return "Un triángulo equilateral con \(anchoDeLado) ancho de lados."
    }
}
var triangle = TrianguloEquilateral(anchoDeLado: 3.1, nombre: "un triangulo")
print(triangulo.perimetro)
triangulo.perimetro = 9.9
print(triangulo. anchoDeLado)
```

En el consultor para `perimetro`, el nuevo valor tiene el nombre implícito de `newValue`. Puedes proveer un nombre explícito en los paréntesis después de `set`.

Nota como el inicializador para la clase `TrianguloEquilateral` tiene tres pasos diferentes:

1. Asignar el valor de las propiedades que la subclase declara.
2. Llamar el inicializador del superclase.
3. Cambiar el valor de propiedades definidos por la superclase. Cualquier trabajo adicional que requiera el uso de métodos, consultores o modificadores, pueden ser usados en este momento.

Si no necesitas calcular el valor de la propiedad, pero aun necesitas proveer código que es ejecutado después de asignar un valor nuevo, usa `willSet` y `didSet`.

El código que proveas es ejecutado cuando el valor cambia afuera de un inicializador. Por ejemplo, el siguiente clase asegura que el ancho de los lados de un triángulo siempre sea igual el ancho de su cuadrado.

```
class TrianguloYCuadrado {
    var triangulo: TrianguloEquilateral {
        willSet {
            cuadrado.anchoDeLado = newValue.anchoDeLado
        }
    }
    var cuadrado: Cuadrado {
        willSet {
            triangulo.anchoDeLado = newValue.anchoDeLado
        }
    }
    init(tamaño: Double, nombre: String) {
        square = Cuadrado(anchoDeLado: tamaño, nombre: nombre)
        triangle = TrianguloEquilateral(anchoDeLado: tamaño, nombre: nombre)
    }
}
var trianguloYCuadrado = TrianguloYCuadrado(tamaño: 10, nombre: "otra figura de prueba")
print(trianguloYCuadrado.cuadrado.anchoDeLado)
print(trianguloYCuadrado.triangulo.anchoDeLado)
trianguloYCuadrado.cuadrado = Cuadrado(anchoDeLado: 50, nombre: "cuadrado más grande")
print(trianguloYCuadrado.triangulo.anchoDeLado)
```

Cuando trabajes con valores opcionales, puedes escribir `?` antes de operaciones como métodos, propiedades, y subíndices. Si el valor antes de `?` es `nil`, todo después del `?` es ignorado y el valor de toda la expresión es `nil`. De lo contrario, el valor opcional es desenvuelto, y todo después del `?` actúa en el valor desenvuelto. En ambos casos, el valor de la expresión es un valor opcional.

```
let cuadradoOpcional: Cuadrado? = Cuadrado(anchoDeLado: 2.5, nombre: "cuadrado opcional")
let anchoDeLado = cuadradoOpcional?.anchoDeLado
```

Enumeraciones y Estructuras

Usa `enum` para crear una enumeración. Como clases y otros tipos nombrados, enumeraciónes pueden tener métodos asociados con ellos.

```
enum Rango: Int {
    case As = 1
    case Dos, Tres, Cuatro, Cinco, Seis, Siete, Ocho, Nueve, Diez
    case Jota, Reina, Rey
    func descripcionSimple() -> String {
        switch self {
        case .As:
            return "as"
        case .Jota:
            return "jota"
        case .Reina:
            return "reina"
        case .Rey:
            return "rey"
        default:
            return String(self.rawValue)
        }
    }
}
let as = Rango.As
let asValor = as.rawValue
```

Glosario

Apple

Apple Inc. es una empresa multinacional estadounidense que diseña y produce equipos electrónicos y software, con sede en Cupertino (California, Estados Unidos). Entre los productos de hardware más conocidos de la empresa se cuenta con equipos Macintosh, el iPod, el iPhone y el iPad.

1.1. Sobre Swift

Arreglo

(Array) En programación también se denomina matriz, vector o formación a una zona de almacenamiento continuo que contiene una serie de elementos del mismo tipo, los elementos de la matriz. Desde el punto de vista lógico una matriz se puede ver como un conjunto de elementos ordenados en fila (o filas y columnas si tuviera dos dimensiones).

1.2. Una Guia de Swift

Booleana

(Boolean, Bool) El tipo de dato lógico o booleano es en computación aquel que puede representar valores de lógica binaria, esto es 2 valores, valores que normalmente representan falso o verdadero.

1.2. Una Guia de Swift

Ciclo

(Loop) Un bucle o ciclo, en programación, es una sentencia que se realiza repetidas veces a un trozo aislado de código, hasta que la condición asignada a dicho bucle deje de cumplirse.

1.2. Una Guia de Swift

Clase

(Class) Una plantilla para la creación de objetos de datos según un modelo predefinido. Las clases se utilizan para representar entidades o conceptos, como los sustantivos en el lenguaje. Cada clase es un modelo que define un conjunto de variables -el estado, y métodos apropiados para operar con dichos datos -el comportamiento. Cada objeto creado a partir de la clase se denomina instancia de la clase.

1.2. Una Guia de Swift

Clausura

(Clousure, Block) Una función evaluada en un entorno que contiene una o más variables dependientes de otro entorno.

1.2. Una Guia de Swift

Cocoa

El marco de trabajo para desarrollo de aplicaciones para la plataforma OS X (antes conocido como Macintosh).

1.1. Sobre Swift

Cocoa Touch

El marco de trabajo para desarrollo de aplicaciones para la plataforma iOS (el sistema operativo de iPhone, iPod y iPad).

1.1. Sobre Swift

Condicional

Una instrucción o grupo de instrucciones que se pueden ejecutar o no en función del valor de una condición.

1.2. Una Guia de Swift

Consultor

(Getter) Método que retorna el valor de una propiedad de un objeto.

1.2. Una Guia de Swift

Conteo de Referencia Automatica

(Automatic Reference Counting, ARC) En Objective-C y programación Swift, Conteo de Referencia Automatica es una mejora de manejo de memoria donde la carga de mantener el conteo de referencia de un objeto es transferido del programador al compilador.

1.1. Sobre Swift

Contexto

(Scope) Es el mínimo conjunto de datos utilizado por una tarea que debe ser guardado para permitir su interrupción en un momento dado, y una posterior continuación desde el punto en el que fue interrumpida en un momento futuro.

1.2. Una Guia de Swift

Depurador

(Debugger) Un depurador, es un programa usado para probar y depurar (eliminar) los errores de otros programas.

1.1. Sobre Swift

Diccionario

(Dictionary) Un diccionario (también contenedor asociativo, mapa, mapeador, hash, vector asociativo, mapa finito, tabla de consulta) es un tipo abstracto de dato formado por una colección de claves únicas y una colección de valores, con una asociación uno a uno.

1.2. Una Guia de Swift

Entrada y Salida

(Input/Output, I/O) Un dispositivo que permite la comunicación entre un sistema de procesamiento de información, tal como la computadora y el mundo exterior, y posiblemente un humano u otro sistema de procesamiento de información.

1.2. Una Guia de Swift

Foundation

El marco de trabajo principal para desarrollo de aplicaciones para todas las plataformas de Apple.

1.1. Sobre Swift

Función

(Function) Se le llama función (también llamada subrutina, subprograma, procedimiento, rutina o método) a un segmento de código separado del bloque principal y que puede ser invocado en cualquier momento desde este u otra.

1.2. Una Guia de Swift

Indice

(Index) El numero asociado con un elemento en un arreglo.

1.2. Una Guia de Swift

Lenguaje Interpretado

(Scripting language) Lenguaje que usa un intérprete envés de un compilador. Los intérpretes se diferencian de los compiladores o de los ensambladores en que mientras estos traducen un programa desde su descripción en un lenguaje de programación al código de máquina del sistema, los intérpretes sólo realizan la traducción a medida que sea necesaria, típicamente, instrucción por instrucción, y normalmente no guardan el resultado de dicha traducción.

1.1. Sobre Swift

Llave

(Key) El texto con que se asocia un valor en un diccionario.

1.2. Una Guia de Swift

Marco de Trabajo

(Framework) Una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos concretos de software, que puede servir de base para la organización y desarrollo de software.

Modificador

(Setter) Método que cambia el valor de una propiedad de un objeto.

Método

(Method) Una función cuyo código es definido en una clase y puede pertenecer tanto a una clase, como es el caso de los métodos de clase o estáticos, como a un objeto, como es el caso de los métodos de instancia. La diferencia entre una función y un método es que éste último, al estar asociado con un objeto o clase en particular, puede acceder y modificar los datos privados del objeto correspondiente de forma tal que sea consistente con el comportamiento deseado para el mismo.

1.2. Una Guia de Swift

Número Entero

(Integer) Un tipo de dato entero en computación es un tipo de dato que puede representar un subconjunto finito de los números enteros.

1.2. Una Guia de Swift

Objective-C

Objective-C es un lenguaje de programación orientado a objetos creado como un superconjunto de C para que implementase un modelo de objetos parecido al de Smalltalk.

1.1. Sobre Swift 1.2. Una Guia de Swift

Opcional

(Optional) Un tipo opcional es un tipo polimórfico que representa la encapsulación de un valor opcional; por ejemplo, se utiliza como el tipo de retorno de las funciones que puedan o no devolver un valor significativo cuando se aplican. Se compone de ya sea un constructor vacío (llamado "Ninguno" o "Nada"), o un constructor que encapsula el tipo de datos original A (escrito "Justo A" o "Algún A"). Fuera de la programación funcional, estos son conocidos como tipos anulables.

1.2. Una Guia de Swift

Orientado a Objetos

La programación orientada a objetos (POO) es un paradigma de programación que usa objetos en sus interacciones, para diseñar aplicaciones y programas informáticos.

1.1. Sobre Swift

Playgrounds

Ambiente de Swift que permite usar el lenguaje como si fuera interpretado y no compilado. Permite programadores experimentar con código Swift y ver los resultados inmediatamente, sin el gasto de compilar y correr la aplicación.

1.1. Sobre Swift

Procedurales

La programación por procedimientos es un paradigma de la programación. Esta técnica consiste en basarse de un número muy bajo de expresiones repetidas, englobarlas todas en un procedimiento o función y llamarlo cada vez que tenga que ejecutarse.

1.1. Sobre Swift

Software

Equipo lógico o soporte lógico de un sistema informático, que comprende el conjunto de los componentes lógicos necesarios que hacen posible la realización de tareas específicas, en contraposición a los componentes físicos que son llamados hardware.

1.1. Sobre Swift

Swift

Lenguaje de programación multiparadigma creado por Apple enfocado en el desarrollo de aplicaciones para sus plataformas.

1.1. Sobre Swift 1.2. Una Guia de Swift 1. Bienvenidos a Swift 0. Introducción

Texto

Conocida como "String" en inglés y la mayoría de lenguajes de programación.

1.2. Una Guia de Swift

Tupla

(Tuple) Conjunto de elementos de distinto tipo que se guardan de forma consecutiva en memoria.

1.2. Una Guia de Swift